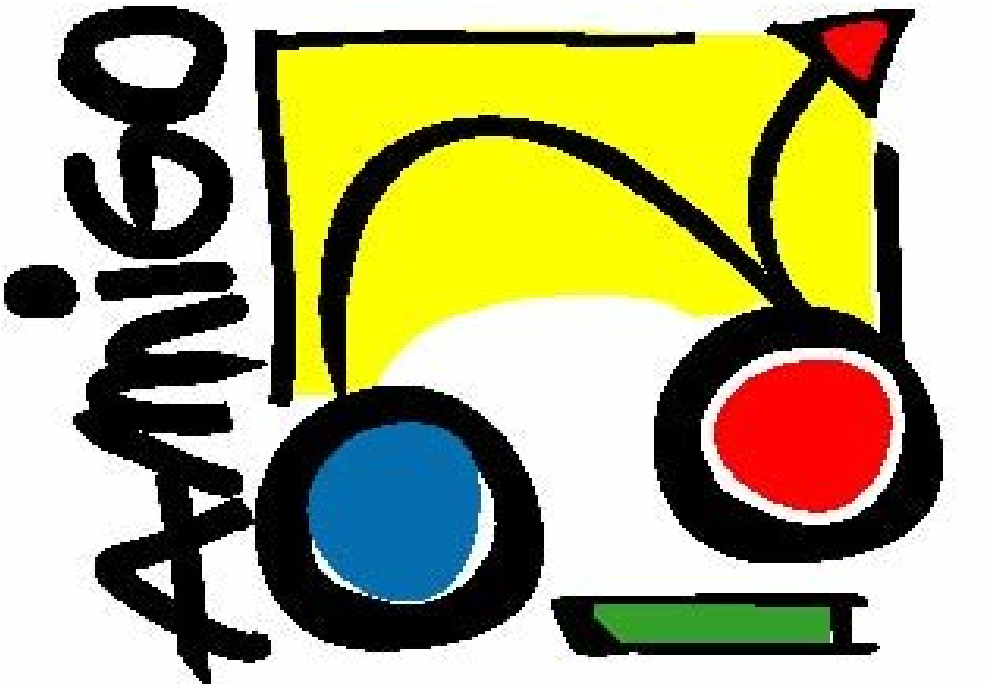

IST Amigo Project
Deliverable D3.3

Amigo Middleware Core Enhanced: Prototype Implementation & Documentation

IST-2004-004182
Public

Project Number	:	IST-004182
Project Title	:	Amigo
Deliverable Type	:	Report + Prototype

Deliverable Number	:	D3.3
Title of Deliverable	:	Amigo Middleware Core Enhanced: Prototype Implementation & Documentation
Nature of Deliverable	:	Public
Internal Document Number	:	amigo-d3.3-final
Contractual Delivery Date	:	31 august 2006
Actual Delivery Date	:	11 October 2006
Contributing WPs	:	WP3
Editor	:	**INRIA:** Nikolaos Georgantas
Author(s)	:	**INRIA:** Sonia Ben Mokhtar, Yérom-David Bromberg, Nikolaos Georgantas, Noha Ibrahim, Valérie Issarny, Wilfried Jouve, Frédéric Le Mouël, Laurent Réveillère, Daniele Sacchetti, Ferda Tartanoglu **FT:** Anne Gérodolle, Mathieu Vallée **ICCS-NTUA:** Miltiades Anagnostou, Ioannis Papaioannou, Ioanna Roussaki, Dimitris Tsesmetzis **IKER:** Jorge Parra **IMS:** Edwin Naroska **Microsoft:** Ron Mevissen, Stephan Tobies **TELIN:** Remco Poortinga, Pravin Prawar, Andrew Tokmakoff **TID:** Jordi García, José María Miranda, Marc Planagumà, Álvaro Ramos, David Roldán **VTT:** Jarmo Kalaoja, Julia Kantorovitch, Ilkka Niskanen, Toni Piirainen

Abstract

D3.3 is the second deliverable on the prototype implementation and associated documentation of most Amigo middleware components, while it also reports on ongoing conceptual and design work for other Amigo middleware components. D3.3 comprises: (i) the present document; (ii) developed source code of components; (iii) developed service description vocabulary and language ontologies; (iv) user's guide and developer's guide documents for components and ontologies; and (v) Javadoc-style and OWLDoc electronic documentation for components and ontologies. Delivered material besides the present document can be accessed – for the moment, in a restricted way – on the Amigo OSS Repository - Public Web

site (http://amigo.gforge.inria.fr/home/index.html). D3.3 addresses: the Amigo programming and deployment framework; service description vocabulary and language; comprehensive service description, discovery, composition, adaptation & execution; interoperable service discovery & interaction middleware; domotic infrastructure; security & privacy; content distribution; data store; accounting & billing; and in-home location management.

Keyword list

ambient intelligence, networked home system, interoperability, mobile/personal computing/consumer electronics/domotic domain, semantic concept, ontology, service description vocabulary, service description language, semantic reasoning, service matching, service composition, service adaptation, service execution, middleware, service discovery protocol, service interaction protocol, programming and deployment framework, context, quality of service, multimedia streaming, content distribution, security, privacy, data storage, accounting & billing, location management

Table of Contents

Figures

Tables

1 Introduction

The present Deliverable D3.3 provides the second official prototype implementation of the Amigo Base Middleware (or simply middleware), after the first prototype implementation delivered by Deliverable D3.2 [Amigo-D3.2].

By title, D3.3 concerns the enhanced prototype implementation and documentation of the Amigo middleware core; nevertheless, as in Deliverable D3.2, we deliver and document herein implementation of both middleware core functionalities and upper middleware functionalities. Further, we report on middleware functionalities for which conceptual and design work is still being carried out. Thus, the level of presentation of different middleware functionalities differs depending on their current stage of progress.

D3.3 covers almost the whole range of the Amigo middleware functionalities, as almost all the WP3 tasks have been active since the delivery of D3.2 (some WP3 tasks specifically started immediately after D3.2). More specifically:

- D3.3 reports on Tasks 3.1, 3.2, 3.4-3.8.
- Task 3.3 completed its objectives and was therefore terminated in Month 24. D3.3 provides final overview report on the components produced by this task.
- Task 3.9 is from now on moved to WP4, due to its affinity with WP4 work and, more particularly, with the context management architecture being elaborated therein. D3.3 provides a – final for WP3 – overview report on the work carried out so far in this task within WP3.
- Task 3.10 has been recently introduced in the DoW; therefore, D3.3 does not cover this task.

Applying the same reporting model as D3.2, D3.3:

- Follows the conventional way of reporting, already employed in the previous deliverables, for ongoing conceptual and design work.
- Follows component-oriented delivering and reporting for implementation work. This, more specifically, includes:
    - As part of the present document, an updated or extended overview with respect to D3.2 for each component under development.
    - On the Amigo OSS Repository - Public Web Site [Amigo-OSS-Pub] (see [Amigo-D9.5]), updates with respect to D3.2 for each component under development:
        - Source code of the current prototype version, if one is already available;
        - User's guide and developer's guide documents, if already available;
        - Javadoc[1] (or equivalent for C#) documentation, if already available.
    - On [Amigo-OSS-Pub], updates with respect to D3.2 for the service description vocabulary and service description language:
        - OWL specification of the current version;
        - User's guide and developer's guide documents, if already available;
        - OWLDoc[2] (follows the same principle as Javadoc) documentation, if already available.

[1] http://java.sun.com/j2se/javadoc/

We note here that, with regard to component/ontology implementation and documentation, our focus during the last 6 months has been on advancing the implementation of components/ontologies; we have, thus, released advanced versions for most of them. Consequently, the online documentation that we provide is mostly still at an early stage and will take a form closer to complete when the almost final, public versions of components/ontologies will be available.

The specific Amigo middleware functionalities, components, and ontologies reported in the chapters of the present document (and, for the latter two, further, when already available, delivered in source along with online documentation) are:

- *Programming and deployment framework*. We provide update with respect to D3.2 (Chapter 2).
- *Service description vocabulary ontologies*. Besides the update with respect to D3.2, we introduce a tool which allows visualizing the semantic, ontology-based descriptions of devices that provide services in the home and associating them with visual representations in their actual locations in the Amigo home (Chapter 3). Thus, this tool supports graphical editing and simulation of home contexts. A first version of the tool has already been developed.
- *Service description – Service discovery, composition, adaptation & execution (SD-SDCAE)*. Based on our previous work on the Amigo-S service description language and performance evaluation of a number of aspects of service discovery, we introduce a comprehensive approach to SD-SDCAE, as all its constituents are very much interrelated (Chapter 4). We further elaborate on two specific constituents of SD-SDCAE, namely, service discovery and service composition.
- *Interoperable service discovery & interaction middleware*. We provide a final overview report of this set of completed components (Chapter 5).
- *Domotic infrastructure*. We provide a final overview report of this set of completed components (Chapter 6).
- *Security and privacy*. We provide update with respect to D3.2 (Chapter 7).
- *Content distribution*. We provide update with respect to D3.2, further introducing the Content Discovery component, which is in charge of discovering new media servers connected to the network and subscribing to their events for possible changes of their contents (Chapter 8).
- *Data store*. We provide update with respect to D3.2 (Chapter 9).
- *Accounting & billing*. We provide an overview of a component supporting accounting & billing for concerned applications in the Amigo home (Chapter 10).
- *In-home location management service*. We provide a – final for WP3 – overview report of this set of components – from now on moved under the responsibility of WP4. This service provides responses to various queries of location-adaptive applications by aggregating inputs from multiple location-determination technologies and services in order to quickly locate devices or users (Chapter 11).

Finally, we conclude with a short discussion on the principal points and progress of the present deliverable (Chapter 12).

[2] http://www.co-ode.org/downloads/owldoc/co-ode-index.php

2 Programming and deployment framework

2.1 Objectives

This chapter proposes a component model that allows clear separation of development and deployment issues. As shown in Section 2.3, the expected result is an “Amigo bundle repository” where components will be available for downloading and installation.

We propose general principles for using a platform like OSGi or .Net, and provide guidelines to developers of functional blocks so that their work can be packed into components that can be further (at deployment time) composed in an arbitrary manner with other components.

The use of this framework is not mandatory, and developers may also package Amigo-aware services as independent applications that are to be deployed on a given system or hardware (as they see fit). Both kinds of components will be able to interact within the same Amigo environment through service discovery protocols (SDPs), communication protocols and (when necessary) interoperability methods.

This chapter is organized as follows: Firstly, we define some vocabulary that we use in the rest of the chapter (Section 2.2), and indicate the expected results of this effort (Section 2.3). We then present the .Net Amigo programming environment (Section 2.4), and, finally, present the OSGi Amigo programming (Section 2.5) and deployment environment (Section 2.6). For both of these environments, this document itemizes a list of components that are either already available or under development. A complete set of component documentation (including a user’s guide, a developer’s guide, and tutorials) is available separately as online documentation referenced at [Amigo-OSS-Pub].

2.2 Vocabulary

Terms like services, interfaces, and components are strongly overloaded in computer science, and – for example – .Net and OSGi use different words to refer to similar concepts, or even the same word to refer to the same concept. In this document, we generally adopt the vocabulary used in OSGi specifications.

In the following:

A *software node* runs on a physical node. It may be a .Net platform, an OSGi platform or any process.

More specifically, a *platform* is a software node where software components can be deployed.

A *functional block* is identified as such in the abstract Amigo architecture: For example, Context Management is a functional block; Security is a functional block, etc.

A *software component* is some part of a functional block that can run on a software node. It communicates with other components of the same functional block using some protocol stack.

A *bundle* is a software component that can be deployed on an OSGi platform. In this section, the word *bundle* is also used to refer to deployable .Net components.

An *interface* defines a set of operations or methods. This word may refer to a programming language-related abstraction (a Java or C# interface can be implemented by a Java or C# class) or to the interface of a remote service, described in an interface description language (IDL). For example, the interface of a Web service is described in WSDL.

A *service* is an artifact provided by a component that offers an interface. It may be:

- A local service: accessible from inside the same software node, in the form of an object implementing a given interface (C# or Java).

- A remote service: accessible from a remote client using a communication protocol through a “service API” (SAPI).

2.3 Expected results

The expected result of this effort is a repository of bundles, which is called the “Amigo Bundle Repository”. Two versions of the repository will be provided, one for .Net bundles, the other for OSGi bundles. Services packed as applications could also be made available on a repository. These repositories will be accessible through HTTP. They will provide for each functional block the list of available bundles; and for each bundle, the documentation of the bundle, the source code of the bundle, and the deployable bundle itself.

The current version of the Amigo OSGi repository is publicly accessible at http://amigo.gforge.inria.fr/obr/v2 (also referenced at [Amigo-OSS-Pub]). Contents of this repository come either from WP3 (see description below) or WP4 (context management-related bundles).

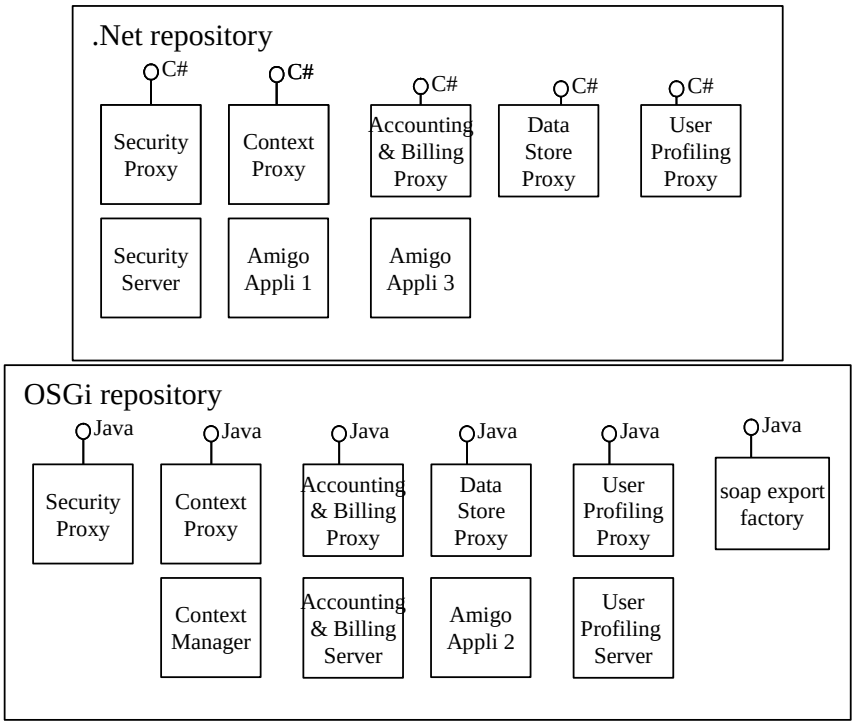

Figure 2-1: The Amigo Bundle Repository contains a set of bundles that can be deployed on a platform (OSGi or .Net). Some bundles provide a Java or C# API that other bundles deployed on the same platform can use. Note: This figure is only illustrative and does not indicate the Amigo bundle repository's final state.

Each functional block may provide one or more bundles that correspond to different parts of the functional block. We can distinguish between three main types of bundles: “server bundles” (e.g., the security server bundle), “proxy bundles” (e.g., the security proxy bundle), and “local bundles” (e.g., the soap export factory).

- Server bundles will be deployed on only one or a few nodes of a network; they will be in general developed for only one of the targeted platforms. Access to services

provided by the server bundles is done through a remote interaction protocol. The word “server” here does not refer to a “client-server” model but simply to the fact that a remote service is offered.

- Proxy bundles should be available for both programming frameworks. They will allow reusability of code among developers of components using a given functional block. Proxy bundles do not provide remote services; but rather a local API that gives access to the distributed functional block by means: either of simple “stubs” (local representatives of remote services working in a client/server model), or of "smart proxies" that offer a simplified view of a distributed system by possibly handling complex interaction. Proxy interfaces allow developers to use a functional block without knowing (at development time) the distributed architecture of the block. In some cases, several implementations of a proxy may be available, and the choice of which implementation to use could be made at deployment time. This is particularly useful when using a complex functional block, where the functionalities offered are clearly identified but the distribution of these functionalities over the network will depend on the network configuration and the capabilities of the nodes.
- Local bundles are not linked to any functional block. They provide services such as logging, protocol adapters, etc.

Figure 2-1 shows how the Amigo bundle repository could look like: if we take the example of the security functional block, on the .Net repository a “server bundle” and a “proxy bundle” may be available, whereas on the OSGi repository only a “proxy bundle” is available. The security proxies offer a simple API to interact with the security functional block. They hide the details of discovering the security server, managing the security protocol, possibly reconnecting to a new security server in case that the current security server becomes unavailable, etc.

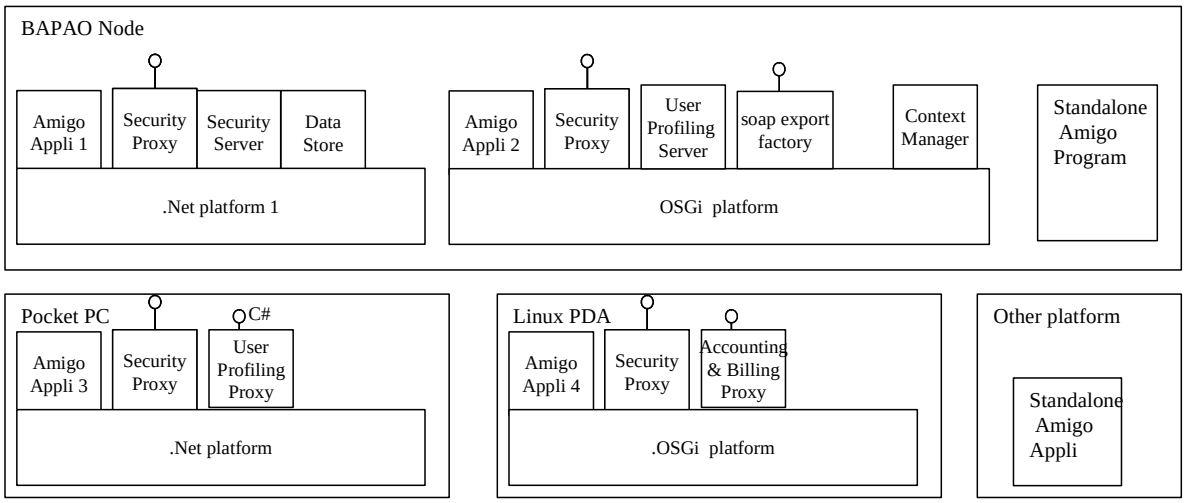

Figure 2-2: An Amigo Network with 4 physical nodes and 6 Amigo software nodes with different configurations. The security proxies discover the security server using WS-discovery and interact with it using SOAP.

Figure 2-2 shows an example of use of the Amigo Bundle Repository shown in Figure 2-1: all server bundles are deployed on a PC (called BAPAO, for “Base Amigo Peripheral that is Always On”), together with some proxy bundles. The security proxy is also deployed on the OSGi platform, for local use by the user profile server proxy. Applications deployed on a Pocket PC or Linux PDA also access the security manager thanks to the locally deployed

security proxy. The standalone Amigo applications may also use the security manager, but they have to manage the complete interaction protocol.

2.4 Amigo .Net programming framework

Provider

Microsoft

Introduction

The .Net (and OSGi) programming frameworks are considered an essential part for an Amigo System since they will be used by nearly all application/component developers as a base.

The goal of the .Net programming framework is to support these developers by enabling them to write their application or component software in a short timeframe by relieving them of time consuming and complex tasks. In this way, developers can concentrate on their core business logic and are not distracted/bothered by complex technologies like remote communication or discovery protocol details.

This programming framework provides developers with a platform on top of the .Net platform that abstracts communication and discovery details from their software. It is almost as if the developer does not need to be concerned about these issues; he/she writes his/her software and in the end incorporates it seamlessly into the programming framework to benefit from its functionalities.

This programming framework will be further extended with common functionalities like logging, configuration, versioning, remote management and software replication mechanisms that are related to deployment. Remote interfaces like those used for configuration and management will be aligned between the programming framework on .Net and the OSGi-based programming framework.

Development status

An initial version was released in M18. Another updated and redesigned version was released in M24 (that includes the compact framework version). The final version will be released in M30.

Intended audience

The programming framework is intended for component as well as for application developers.

License

See EMIC license (Annex A).

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

This component is an SDK and, as such, part of a developer's environment (e.g., IDE). The source software can be compiled with any version of Visual Studio 2005. The library can be used from any development environment with a .Net or .NetCF 2.0 language (e.g., Visual Basic, Visual J++ etc.)

Platform

Microsoft .Net 2.0 / Microsoft .NetCF 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

See [Amigo-OSS-Pub]

Documents

Developer's guide: See [Amigo-OSS-Pub]

User's guide: See [Amigo-OSS-Pub]

Tasks

Final release in M30.

Bugs

Fixed in the latest (M24) version:

Internal #	Description
380	A MessageTransceiver is created in the constructor of DiscoverableService. If this fails (no network interface available -> sockets can not be bound), an exception is thrown which is caught in the constructor of DiscoverableService. This however leads to a partially created class (e.g., MetaData is not created). This behavior (no network->messagetransceiver cannot be instantiated) is transparent to the user, but he/she suddenly has to deal with a partially intitialized class. Constructors should not throw exceptions. It would be better to move this functionality to an Initialize() method returning true or false.
381	DiscoverableService was derived from MarshalByRefObject. This should be removed since there is no reason why a DiscoverableService should cross AppDomains.
383	The option setting (drop membership) could throw an exception if the network is not available at that time. Dispose methods should NOT throw any exceptions.
387	AppSequence is never deserialized due the IsEmptyElement() test. An AppSequence is ALWAYS empty since it only contains attributes (and the XmlReader considers an element with only attributes (but no content) still an empty element).
410	Add a copy constructor to ServiceInfo so that users can create copies of this object.
411	Add constructor for DiscoverableService with Types, Scopes, Location and Endpointrefence, and remove the delegate mechanism.

Patches

Not yet available

2.5 Amigo OSGi programming framework

2.5.1 Context

An OSGi platform allows deployable elements, called "bundles", to be remotely installed from any URL, e.g., from HTTP servers. A bundle is a jar file containing Java code, a special manifest describing the bundle's capabilities, and possibly other resources. When started, a bundle can provide "services". In OSGi terminology, a service may be any Java object. OSGi platforms provide a service registry which allows:

- Registering an object as a (local) service, which means associating this object with a list of properties described in an LDAP syntax, among which is the provided Java interface(s).
- Look up services matching target criteria.

Additionally, the OSGi framework takes care of the life-cycle of services, and automatically suppresses the references of services registered by a bundle when this bundle is stopped. As any Java object can be registered as an OSGi service, Amigo APIs developed in Java can easily be provided as OSGi services and packed in OSGi bundles.

Many useful OSGi bundles are already available on the Web. Here, we briefly introduce some open source bundles that the Amigo OSGi programming and deployment framework uses:

- The Oscar Bundle Repository[3] bundle allows accessing a set of OSGi bundles on a repository accessible through HTTP. When installing a new bundle, the OBR bundle will take care of dependencies and install (if necessary) bundles that provide packages needed by this bundle.
- The Service binder[4] provides an XML language to declare services offered and required by a Java component. Service binder is now standardized in OSGi R4[5] as "declarative services".
- Oscar[6] provides an implementation of the standard OSGi HTTP service, which allows servlet deployment on an OSGi platform. This will be the base to provide Amigo services as Web services.
- The domoware[7] UPnP base driver implements the UPnP base driver specification standard defined by OSGi.
- Knopflerfish[8] has packed the Axis[9] servlet into a bundle. When the Axis bundle is running, objects registered with the OSGi lookup with the property "SOAP.service.name" set are automatically made available as Web services.

2.5.2 Description of work

The Amigo OSGi programming framework includes standard or legacy OSGi bundles, as those described above. The work related to OSGi in Task 3.4 will consist of:

- Maintaining the Amigo OSGi Bundle Repository – help partners to pack Java components in the form of an OSGi bundle and make them available on the repository.
- Provide additional bundles to ease the development of distributed services. This is described in the following sections. Section 2.5.3 introduces the main principles; Section 2.5.5 gives an example of code using this environment; and Section 2.5.6 details the subcomponents that are already available or are planned.
- Provide enhanced tools that ease the deployment of Amigo bundles according to semantic criteria. This is described in Section 2.6.

2.5.3 Components aimed to ease the development of distributed services

We further rely on the fundamental concepts of "export factories" and "binding factories". An "export factory" is a service that makes a Java object remotely available. For this purpose, an

[3] http://oscar-osgi.sourceforge.net/

[4] http://gravity.sourceforge.net/servicebinder/

[5] "OSGi Service Platform, Release 4 CORE" , http://www.osgi.org/osgi_technology/

[6] http://oscar.objectweb.org/

[7] http://domoware.isti.cnr.it/

[8] http://www.knopflerfish.org/

[9] http://ws.apache.org/axis/

export factory provides a method (called "export"). The result of "Exporting a service" is an "Amigo reference" that can be serialized and published using a discovery protocol. This "Amigo reference" contains all useful information to allow a client to access the service, such as the host name and port number where the service can be found, the communication protocols that can be used, etc. Exporting a service may or not involve the construction of some dedicated objects on the server. Symmetrically, a "binding factory" is used on the client to access a given service, given an "Amigo reference". A binding factory provides a method that takes an Amigo service description as parameter and returns a "stub". This stub can then be used by the client to communicate with the remote object. Export factories, binding factories and SDP implementations are packaged in OSGi bundles as follows:

- The Amigo core bundle provides Java interfaces representing the export factory, binding factory, and lookup abstractions; together with basic mechanisms which allow clients to export an object by using (in a transparent way) the currently deployed export factory, or to build a stub to connect to a remote service.
- Specialized bundles (e.g., the kSOAP export bundle, the SLP bundle) provide implementations of these interfaces based on a specific protocol and a specific technology.

Programmers of Amigo-aware bundles that use this framework only need to know the interfaces defined in the Amigo core bundle. The choice of the underlying protocol that an Amigo-aware bundle uses is done at deployment time and depends on the specialized bundles that are deployed together with this Amigo-aware bundle.

A subset of these bundles will be installed on every OSGi node, depending on which type of application bundles the node will host, and the capacities of the hardware platform. It may be desirable to limit the memory footprint on embedded devices. Furthermore, a specific protocol may be preferred depending on the network configuration: in some circumstances, the HTTP protocol may be preferred because of firewall problems, whereas, for communication between Java nodes, JRMP (Java Remote Method Protocol) may be preferred for performance reasons. Therefore, an OSGi platform running on a PDA and hosting only client applications could host only binding bundles, and be limited to a single binding technology (e.g., kSOAP), whereas a platform running on a PC and hosting a variety of server and client applications would host several export and binding factories, so as to maximize interoperability with other nodes.

The proposed approach facilitates the introduction of new protocols, as this involves only writing the corresponding export and binding factories, and packing those as OSGi bundles that register the factories as services. These bundles can then be installed on already existing OSGi nodes, and provide the possibility for already installed applications to export their services or access services using this new protocol. This method makes it possible to expose a service through several protocols, keeping the overhead for the service programmer as lightweight as possible. Exposing the same service according to various protocols reduces the need for translation services and increases communication efficiency. A "client" can access a service running on a remote OSGi platform, provided there is a binding factory running on the client's OSGi platform that is compatible with one of the export factories used on the server's OSGi platform. However, in the case of incompatible binding/export factories (e.g., an embedded server that would provide only a SOAP export service and an embedded client that would contain only a RMI binding service), interoperability methods developed inside Amigo Task 3.3 will be used. Note that interoperability methods may themselves be packed as bundles and deployed on an OSGi platform.

2.5.4 Changes since Deliverable 3.2

In the first version of the OSGi framework, the discovery was made using SLP (Service Location Protocol). At present time, the choice has been made to support WS technologies:

the WS-Discovery protocol for service publication/lookup, the HTTP/SOAP protocols for synchronous communication, and the WS-Eventing protocol for asynchronous communication.

Subsequently, priority was given to the integration of WS-Discovery and WS-Eventing protocols, with focus on interoperability with the .Net Amigo framework. No Java implementation of WS-Discovery being at this time available, it was developed from scratch within Amigo. Concerning WS-Eventing, although the use of this protocol was not foreseen in the previous working plan, we decided to support it in order to facilitate the development and interoperability of Amigo applications. The open source Apache/Jakarta "pubscribe" project supports WS-Eventing. However, we estimated the effort needed for integrating this implementation in our environment higher than that of developing the minimal subset useful for Amigo needs.

Resources initially dedicated to other developments were put on the previously mentioned points. This explains why some bundles that were planned for Month 24 or before are not yet available or not fully functional. The UPnP Amigo bundle (initially planned for Month 24) will be delivered in Month 26. In addition, some unexpected problems were encountered when trying to integrate the Axis libraries into our environment. A first version of the Amigo Axis bundles was delivered in Month 22 (initially planned for Month 20), but several problems with it remain.

Finally, the kSOAP bundle mentioned in the previous deliverable has been suppressed. The corresponding libraries are now embedded in the amigo_ksoap_binding bundle.

2.5.5 Getting started with the OSGi framework

Apart from the Java documentation, the most useful documents for people interesting in using this framework are three small tutorials aimed to help the developer getting started with the OSGi framework. Each of them is associated with a sample Java project that can be downloaded and used as a basis for new bundle development.

- http://amigo.gforge.inria.fr/obr/tutorial/tutorial_v1.htm. The first part of this document is not specifically aimed at developers. It helps downloading and installing an OSGi platform (in this case Oscar), and deploying and running Amigo bundles on this platform so as to launch a client or server application. The second part shows how to develop a new bundle using the Amigo core interface and working with "generic stubs".
- http://amigo.gforge.inria.fr/obr/tutorial/tutorial_v2.htm. This tutorial helps the developer using new features related to Java code generation.
- http://amigo.gforge.inria.fr/obr/tutorial/tutorial_wsevent.htm. This tutorial helps the developer in writing a distributed application with an event source and an event subscriber.

The following code excerpts illustrate the API offered to the programmer for publication/ discovering of services and synchronous communication.

2.5.5.1 Writing an Amigo service

We suppose here that a developer writes a Java class (HelloImpl) that implements some Java interface (Hello). This developer wishes to make the object available on the network as a *service*, using the default Amigo communication and discovery protocols.

Hereafter is the code of this component:

```
1   Public class HelloImpl implements Hello{
2       //implement the Hello interface
3       Public String sayHello(String argument){
4           …..
5       }
6       // define fields that reference the middleware Amigo components
7       AmigoLdapLookup lookup;
8       ServiceExporter serviceExporter;
9       // define methods that set these fields
10      public void setLookup(AmigoLdapLookup lookup){
11          this.lookup=lookup;
12      }
13      public void unsetLookup(AmigoLdapLookup lookup){
14          if (lookup==this.lookup) lookup=null;
15      }
16      public void setServiceExporter(ServiceExporter serviceExporter){
17          this.serviceExporter=serviceExporter;
18      }
19      public void unsetServiceExporter(ServiceExporter serviceExporter){
20          if (serviceExporter==this.serviceExporter) serviceExporter=null;
21      }
22
23      public void activate(){
24          // 1- create an instance of “AmigoService” that describes this object
25          AmigoService service = serviceExporter.createService(this);
26          // 2- create an “exported reference” so that this object is accessible through
    the default remote protocol (e.g. SOAP)
27          service.exportMethods(AmigoReference.DEFAULT, Hello.class);
28          // 3- advertise this reference as a “Hello” service with some additional
    property called nodeName
29          service.addProperty(“serviceType”,”Hello”);
30          String nodeName = System.getProperty(“nodeLocation”);
31          service.addProperty(“nodeLocation”,nodeName);
32          lookup.register(service);
33      }
34  }
```

In this example, the HelloImpl class uses the service binder to find instances implementing the ServiceExporter and AmigoLdapLookup interface. To that purpose, the developer has defined 2 fields (lines 7 and 8) and written methods that set this field (lines 10 to 21). He/she defines some metadata (shown below) that describe the dependencies of HelloImpl (it needs a ServiceExporter and Lookup instance to work properly), and then packs this class into an OSGi bundle together with a service binder activator.

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="com.francetelecom.amigo.hello.HelloImpl">
    <requires service="com.francetelecom.amigo.core.AmigoLdapLookup"
      filter=""
      cardinality="1"
      policy="dynamic"
      bind-method="setLookup"
      unbind-method="unsetLookup"
    />
    <requires service="com.francetelecom.amigo.core.ServiceExporter"
      filter=""
      cardinality="1..n"
      policy="dynamic"
      bind-method="setServiceExporter"
      unbind-method="unsetServiceExporter"
    />
  </component>
</bundle>
```

This metadata describes that the component needs an instance of AmigoLdapLookup and at least one instance of ServiceExporter.

When deployed, the service binder will create an instance of HelloImpl and create an instance manager for this component. This instance manager is in charge of calling the setLookup and setServiceExporter method when the lookup and service exporter will be available. Once both dependencies are resolved, the activate method is called. The service is exported (lines 25-27), i.e., a reference allowing to access this object remotely, e.g. a HTTP URL, is created; and then registered with SDP with two properties, serviceType and nodeLocation (lines 29-32). Line 27 indicates that all methods defined by the Hello interface (i.e., sayHello) must be made remotely accessible.

2.5.5.2 Discovering and using a service

The developer now wants to write a service that needs to access an instance of Hello service. For that purpose, he/she writes a HelloUser class.

```
1  Public class HelloUser {
2  // defines a field that references the Amigo Lookup
3  AmigoLdapLookup lookup;
4  // define methods that set these fields
5  public void setLookup(AmigoLdapLookup lookup){
6     this.lookup=lookup;
7  }
8  public void unsetLookup(AmigoLdapLookup lookup){
9     if (lookup==this.lookup) lookup=null;
10 }
11
12 public void activate(){
13    // 1- find which “Hello” services are available on the network
14    String scope = “serviceType=Hello”;
15    AmigoService[] services = lookup.lookup(request);
16    // 2- invoke all Hello services
17    for (int i=0;i<services.length;i++){
18         // print out the location of the service
19         System.out.println(“found a hello service running at location”+
20            Services[i].getProperty(“nodeLocation”);
21         try{
22            // get a stub
23            Stub stub=services[i].getSpecificStub(Hello.class);
24            String result = stub.sayHello(“World”);
25            System.out.println(“this service answers “+result);
26            }catch(AmigoException ex){
27                 System.err.println(“impossible to create a stub for
Reference “+services(i).getReference());
28            }
29    )
30 }
31}
```

The HelloUser class also relies on the service binder to discover the instance of the lookup middleware component. Hereafter is the metadata of this component:

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="com.francetelecom.amigo.hello.HelloUser">
    <requires service="com.francetelecom.amigo.core.AmigoLdapLookup"
      filter=""
      cardinality="1"
      policy="dynamic"
      bind-method="setLookup"
      unbind-method="unsetLookup"
    />
  </component>
</bundle>
```

When activated, the client looks for all services that have been published with a serviceType named Hello (line 14-15). It iterates on all found elements (line 17). For each of them, it tries to build a stub implementing the Hello interface (line 23). Then it calls the “sayHello” method of this stub (line 24), which in turn results in calling the corresponding “sayHello” method of the remote server.

2.5.5.3 Deploying the HelloImpl and HelloUser components

The HelloImpl and HelloUser components may be deployed on any node of the network that runs an OSGi platform. When activated, HelloUser will discover all instances of HelloImpl and call the sayHello method. The only conditions are that: HelloImpl is deployed together with (at

least) a bundle providing an ExportFactory service and a bundle providing an AmigoLdapLookup service; and HelloUser is deployed together with (at least) a bundle providing an AmigoLdapLookup service and a bundle providing a BindingFactory service able to handle the references created by the export factory used by HelloImpl.

2.5.6 List of Amigo OSGi bundles

As stated before, the OSGi-based programming & deployment framework is composed of a series of OSGi bundles. A subset of these bundles or all these bundles may be installed on each OSGi platform of an Amigo system. The set of bundles installed on an OSGi platform determines the Amigo profile of this platform.

The OSGi bundles presented herein may belong to different categories:

- Encapsulation of OSS libraries developed in another open source project. Then, the license terms should be the same as those of the original project.
- Original Amigo bundles. The license chosen by France Telecom for these bundles is LGPL. These bundles include an Amigo core bundle (which provides interfaces and basic mechanisms) and specialized bundles that provide adaptation to different protocols.

These bundles are available on the Amigo OSGi bundle repository, which contains: (i) binary bundles ready for deployment on an OSGi platform, (ii) documentation associated to these bundles, and (iii) source code corresponding to the binary release. The source code is also available on the Amigo source code management repository [Amigo-OSS-SCM].

The current Amigo OSGi bundle repository is available at http://amigo.gforge.inria.fr/obr/v2 (also referenced at [Amigo-OSS-Pub]).

Table 2-1 lists the sub-components of the OSGi framework and their dependencies.

Component	Type	Depends on	License	Availability	Changes
log4j	Ext. library		Apache	Month 20	
Amigo core	Amigo	Log4j	LGPL	Month 20	
Amigo kSOAP binding factory	Amigo	Amigo core	LGPL	Month 20	see below
Amigo kSOAP export factory	Amigo	kSOAP binding factory, Amigo core, OSGi HTTP service	LGPL	Month 20	see below
Amigo Axis export factory	Amigo	Amigo core, Axis, OSGi HTTP service	LGPL	Month 22	see below
Amigo Axis binding factory	Amigo	Amigo core, Axis,	LGPL	Month 22	see below
Amigo SLP adapter	Amigo	Log4j, Amigo core	LGPL	Month 20	see below
Amigo UPnP adapter	Amigo	Log4j, Amigo core, OSGi UPnP base driver	LGPL	Month 26	initially planned Month 24
Amigo WS-discovery adapter	Amigo	Log4j, Amigo core, kSOAP	LGPL	Month 21	initially planned Month 24
Amigo Service Binder	Amigo	Log4j, Amigo core	LGPL	Month 27	
Amigo Semantic adaptation bundles	Amigo	Log4j, Amigo core	LGPL	Month 30	

Table 2-1: Sub-components of the OSGi framework

2.5.6.1 log4j Bundle (Library Bundle)

Provider

Library provided by Apache / OSGi encapsulation by France Telecom

Introduction

This bundle encapsulates the log4j library, an open flexible logging system for Java applications. log4j is developed within the Apache project (http://logging.apache.org/log4j).

Development status

Done: encapsulation of log4j 1.2.13

First release done in Month 18 for Amigo partners

Public release done in Month 20

Intended audience

This is general purpose software for any Java developer.

License

Apache license

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

- log4j.jar contains the log4j bundle
- test-log4j.jar contains a bundle that uses the log4j bundle

Documents

For general documentation see http://logging.apache.org. The OSGi bundle is provided with an example of use.

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.2 Amigo Core OSGi Bundle

Provider

France Telecom

Introduction

This bundle provides the Java interfaces and core classes that form the Amigo programming framework core: interfaces ExportFactory, BindingFactory, AmigoLdapLookup, etc.

Remark: This bundle provides basic mechanisms for communication and service discovery, but is not linked with any protocol. It should be deployed together with implementing bundles related to communication protocols (binding factories and/or export factories) or service discovery protocols.

Development status

First version was made available in Month 18 for Amigo partners.

This initial version has been augmented (public release in Month 20) in order to define abstractions that describe asynchronous eventing mechanisms on one hand, dynamic stub generation on the other hand.

Intended audience

- Java developers that want to expose Java objects as remote services or event sources;
- Java developers that want to access to remote services or subscribe to events;
- Java developers that want to write an adapter for a given technology.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

The bundle appears on the bundle repository under a “bundle name” indicated in brackets.

- amigo_core.jar: bundle that provides the core interfaces and classes (bundle name amigo_core);
- hello_server.jar: test bundle that exports a simple Hello service (bundle name amigo_test_hello_server);
- hello_client.jar: test bundle that uses a Hello service using a well-known endpoint (bundle name amigo_test_hello_client);
- hello_lookup_client: test bundle that discovers the available Hello services and uses the first discovered (bundle name amigo_test_hello_lookup_client);
- test_pictureframe_server.jar: test bundle that provides a “picture frame” (bundle name amigo_test_pictureFrame_server) as an Amigo service.
- test_pictureFrame_client.jar: test bundle for the Amigo test picture frame server: this displays a graphical interface to choose an image from available images on the client’s file system to be displayed by the “picture frame” server (bundle name amigo_test_pictureFrame_client).

Documents

Java documentation, tutorial, developer’s and user’s guides are available on http://amigo.gforge.inria.fr/obr/ (also referenced at [Amigo-OSS-Pub]).

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.3 Amigo kSOAP Binding Factory Bundle

Provider

France Telecom

Introduction

This bundle allows building a stub to a remote object accessible through the SOAP protocol. It provides a local OSGi service that implements the BindingFactory interface.

It also takes into account the client API for WS-Eventing, allowing a client to subscribe from an event source or unsubscribe and asynchronously receive events.

Development status

Initial release was made available in Month 18 for Amigo partners.

Several changes have been made (public release in Month 20) since the initial release:

- This bundle now encapsulates the kSOAP2 and kxml2 libraries, which allow writing XML- or SOAP-related applications for any Java target (Midp, personal profile, J2SE). kSOAP2 is developed within the kObject project (http://kobject.sourceforge.net); kXLM2 is developed within the kXML project (http://kxml.sourceforge.net/). In the initial OSGi framework release, these two libraries were provided in a separate bundle.
- "Complex" arguments or return types are now taken into account, up to a certain extent: "Valid objects" are either primitive types, hash tables containing primitive types or objects containing fields which are "valid objects". Objects are passed by values. No reference cycle is authorized.
- Tools for stub generation and WSDL parsing are available for the programmer.
- This bundle now implements the WS-Eventing protocol, in a way compatible with the .Net framework implementation.

Intended audience

Network administrators who want to use HTTP/SOAP and WS-eventing as the base communication protocols should deploy this bundle on every OSGi platform that will access remote services or provide a remote service using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java personal profile or J2SE

Software: log4j bundle

Platform

Java (personal profile or J2SE), OSGi

Tools

The sample project related to the Amigo V2 tutorial includes two utilitary tools:

- Java2Stub: parses a Java interface and generates additional files useful both on server and client side.
- WSDL2Java: a variant of the WSDL2Java tool from Axis, which can be used to generate Java interfaces from an existing WSDL description.

Files

amigo_ksoap_binding.jar

Documents

Java documentation

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.4 Amigo kSOAP Export Factory Bundle

Provider

France Telecom

Introduction

This bundle allows making a Java object available through the SOAP protocol. It provides a local OSGi service that implements the ExportFactory interface. The choice of the kSOAP library allows this bundle to be deployed on constrained devices.

Development status

Initial release was made available in Month 18 for Amigo partners.

Public release was made available in Month 20.

Intended audience

Network administrators who want to use HTTP/SOAP as the base communication protocol should deploy this bundle on every OSGi platform that will provide remote services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java personal profile or J2SE

Software: kSOAP binding factory bundle, log4j bundle, HTTP service, servlet

Platform

Java (personal profile or J2SE), OSGi

Tools

ant

Files

amigo_ksoap_export.jar

Documents

Java documentation

Tasks

None

Bugs

None

Patches

None

2.5.6.5 Axis Export Factory Bundle

Provider

France Telecom

Introduction

This bundle allows making a Java object available as a Web service. It provides a local OSGi service that implements the ExportFactory interface.

Development status

It was released in Month 22. As some interoperability problems have been detected, it is recommended to use the kSOAP implementation instead.

Intended audience

Network administrators who want to use HTTP/SOAP as the base communication protocol and provide WSDL service descriptions may deploy this bundle on every OSGi platform that will publish Java object as Web services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE

Software: HTTP service, servlet, axis bundle provided by Knopflerfish

Platform

Java (J2SE), OSGi

Tools

None

Files

amigo_axis_binding.jar

Documents

Java documentation

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.6 Axis Binding Factory Bundle

Provider

France Telecom

Introduction

This bundle allows accessing a Web service. It provides a local OSGi service that implements the BindingFactory interface.

Development status

First version was released in Month 22. There are still misbehaviors, and some interoperability problems have been detected. The Knopflerfish project is no more maintaining the Axis bundle. It is recommended to use the ksoap bundles instead.

Intended audience

Network administrators who want to use SOAP/WSDL as the base protocol for communication/service description may deploy this bundle on every OSGi platform that will access to Web services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE

Software: axis bundle provided by Knopflerfish

Platform

Java (J2SE), OSGi

Tools

None

Files

amigo_axis_export.jar

Documents

Java documentation

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.7 SLP Bundle

Provider

France Telecom

Introduction

This bundle provides an implementation of AmigoLdapLookup based on SLP (Service Location Protocol).

Development status

Initial version (Month 20) based on mesh SLP (Columbia University) under test. Original library can be found at http://mslp.sourceforge.net/ .

No improvements of this bundle have been made, since Amigo partners have agreed to use WS-Discovery as the reference protocol for discovery.

Intended audience

Network administrators who want to use SLP as the base protocol for service discovery in Amigo may deploy this bundle on every OSGi platform that will access to SLP using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE or personal profile

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_meshslp.jar

Documents

Java documentation

Tasks

None

Bugs

Misbehaviors are observed when a large number of platforms using this bundle are running on the same network (in particular on WiFi).

This bundle is no longer maintained, as the WS-Discovery protocol will be used within Amigo.

Patches

None so far

2.5.6.8 UPnP bundle

Provider

France Telecom

Introduction

This bundle will provide an implementation of AmigoLdapLookup based on UPnP.

Development status

First release was initially foreseen for Month 24.

Release has been delayed to Month 26: we need to decide about the mapping of UPnP models on Amigo Service abstractions.

People willing to use UPnP for communication with legacy devices / control points should preferably use implementations of the OSGi UPnP base driver specification, which provides a complete mapping of the UPnP model on OSGi.

Intended audience

Network administrators who want to use UPnP as the base protocol for service discovery may deploy this bundle on every OSGi platform that will access UPnP using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software: any implementation of OSGi UPnP base driver

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

None

Bugs

None so far

Patches

None so far

2.5.6.9 WS-Discovery Bundle

Provider

France Telecom

Introduction

This bundle provides an implementation of AmigoLdapLookup based on WS-Discovery.

Development status

First release made available in Month 21 for Amigo partners

Intended audience

Network administrators who want to use WS-Discovery as the base protocol for service discovery may deploy this bundle on every OSGi platform that will access to WS-Discovery using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software : not yet known

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

None

Bugs

Under correction: bugs related to the use of distinguished names in service types

Patches

4 versions have been delivered since the initial release 1.0.0, which handled only probe and probe match messages. The current 1.0.4 version takes into account resolve/resolvematch messages and discovery proxy, contains some bug corrections, and allows clients to use distinguished names in service types.

2.5.6.10 Amigo Service Binder

Provider

France Telecom

Introduction

The OSGi “declarative services” (formerly, service binder) allows to automatically manage the dependencies between services on the same OSGi platform, by defining a declarative language to describe dependencies and by providing a bundle that instantiates service objects and manages dependencies using the OSGi discovery service. The Amigo Service binder will extend this abstraction to distributed services discovered through a Service Discovery Protocol.

Development status

First release in Month 27 for Amigo partners

Intended audience

Developers who provide services that depend on other services.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software: not yet known

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

First release in Month 27

Bugs

None so far

Patches

None so far

2.5.6.11 Semantic Adaptation Bundles

Provider

France Telecom

Introduction

These bundles will provide mechanisms for adaptation between a client requiring a service and a server providing a service "close enough" to that required by the client. They will provide the following functionality:

- Dynamic translation of component interfaces based on service matching description

These bundles will ease the use of enhanced service discovery for OSGi programmers. They will highly depend on the Service description - Service discovery, composition, adaptation & execution comprehensive approach introduced in Chapter 4.

Development status

Semantic adaptation bundles will be available in Month 30. At present time, a bundle called "Amigo_stubgen" has been developed as a first preliminary step. This bundle is able to dynamically generate a stub class that implements a given Java interface, assuming a direct mapping between this interface and the provided service. The stub bytecode is generated on the fly using the bcel library[10].

Intended audience

Application service developers that seek to dynamically discover and use heterogeneous services available in the environment.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java

These bundles will relate to the Service description - Service discovery, composition, adaptation & execution approach.

Platform

Java (version still to be determined), OSGi

Tools

None

[10] http://jakarta.apache.org/**bcel**/

Files

Not yet available

Documents

Not yet available

2.6 Amigo OSGi deployment framework

The goal of the Amigo OSGi Deployment Framework is to provide a Dynamic Service Deployment functionality that takes into account the semantic description of services and the semantic description of the deployment itself to apply a semantically and timely local deployment strategy.

Usually, the deployment of services is decided statically after the development of the application. Before running an application, its components are deployed in an unchanging way. The innovation of our approach is that it provides dynamic deployment that goes along with the dynamic nature of the environment and semantic deployment that takes into account the nature of devices/platforms and the nature of the context present at a time being.

The semantic deployment functionality is provided by the Dynamic Service Deployment service (see

Figure 2-3). Our service interacts with other high-level services of the middleware that come from the Service description - Service discovery, composition, adaptation & execution comprehensive approach introduced in Chapter 4 (e.g., Enhanced Service Discovery, Service Matching Tool). These high-level services rely on more classic services, the Discovery Service and the Interoperability Service for the discovery and the protocol transformation. Communication interfaces of these services are defined by a service API (SAPI) family, which is a set of possible interfaces such as Amigo interfaces or legacy interfaces (UPnP, etc.).

The internal architecture of the Dynamic Service Deployment service is as follows:

- Service Container: the Service Container stores current services executing on the current platform. This container can be filled locally by the current platform or remotely by other Dynamic Service Deployment services.
- Deployment Strategy: the Deployment Strategy is in charge of deciding the deployment target, i.e., a software node on a remote host, and also the duration of the deployment.

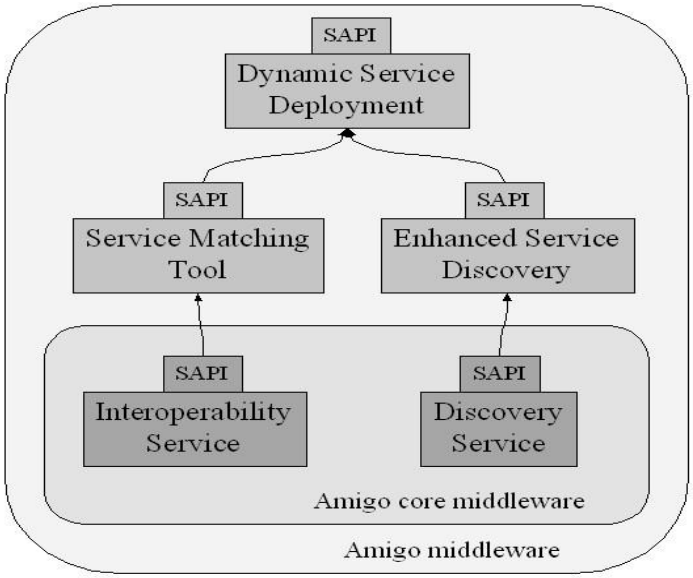

Figure 2-3: Dynamic Service Deployment service in the Amigo Middleware

2.6.1.1 Dynamic Service Deployment service

Provider

INRIA

Introduction

The Dynamic Service Deployment service will offer two functionalities. First, uploading a service using its reference or a semantic description from a specific URL or from an environment description. Second, downloading a service into a context using its reference or semantic description.

Development status

Not yet available. Development started in 2006.

Intended audience

The deployment framework is intended for component as well as for application developers.

Licence

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_dynamicservicedeployment.jar

Documents

Java documentation

Component-specific documentation not yet available

Tasks

Semantic deployment bundles will be available in M30.

Bugs

None so far

Patches

None so far

3 Service description vocabulary ontologies

3.1 Introduction

The objective of this chapter is to provide the latest updates on the work performed in the scope of Task 3.1 on vocabulary ontologies modelling. The chapter is structured as follows. First, the overview of the management of vocabularies from the developer's and also the user's point of view is given (Section 3.2). Then, the ontology visualization tool supporting the service developer's work is introduced (Section 3.3). Next, the main modifications to vocabulary ontologies in the iteration of the present deliverable are presented (Section 3.4). Finally, some examples of using the developed ontologies are provided (Section 3.5).

3.2 Management of vocabularies during development

Figure 3-1 shows how the management of vocabularies during development has been organized. The GForge repository [Amigo-OSS-SCM] provides a version management operation that supports collaboration and sharing of development work between partners. The latest release of vocabularies is provided on the public Web site that will be accessible by the vocabulary developers, vocabulary users and Amigo applications [Amigo-OSS-Pub]. During the modelling, the vocabularies can be imported directly from the vocabulary Web site. If needed, it is also possible to use a local version of a specific vocabulary for testing.

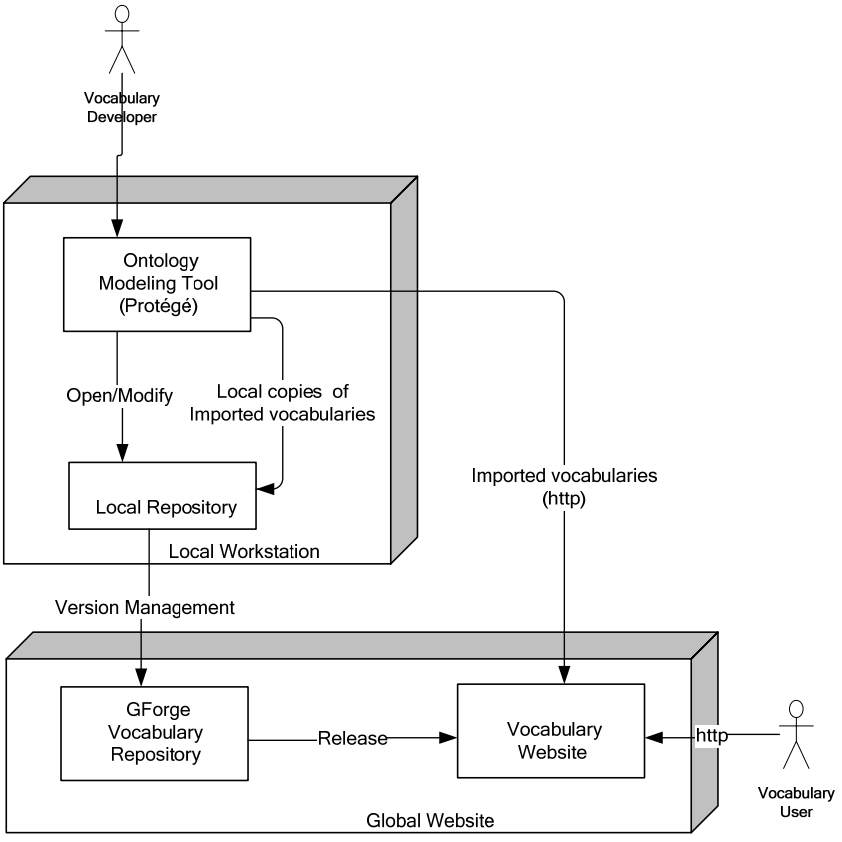

Figure 3-1: The management of vocabularies

3.3 Tool support for vocabulary users

Ontologies can be very complex, and looking at an OWL ontology in XML for the first time can be overwhelming. The gap from beginner to intermediate OWL ontology reader is cumbersome. There are a good number of tools available for ontology development and visualisation, but these require at least a basic understanding of semantic models to begin with. Even for a person with a background on the semantic models and development tools it is often difficult to grasp the overall picture of all vocabularies related to the Amigo home.

This gap could be shortened by visualizing the semantic descriptions of devices providing services and by associating those with visual representations in their actual locations in the Amigo home. The idea of the proposed tool is presented in Figure 3-2.

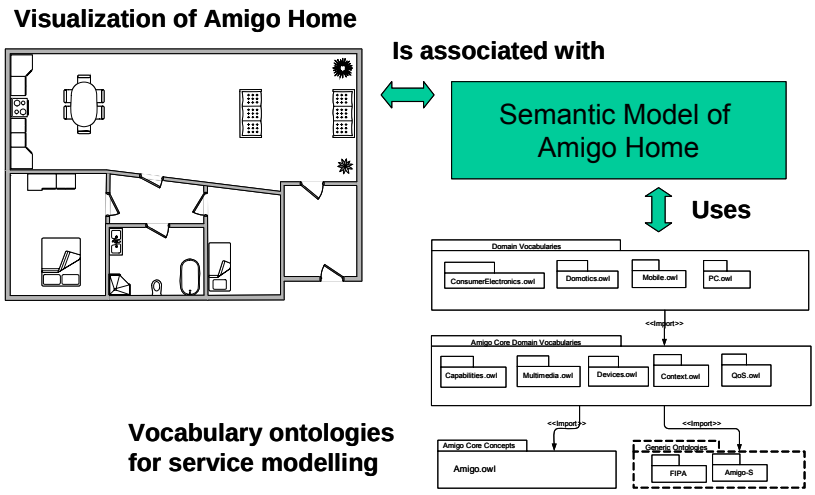

Figure 3-2: Vocabulary visualization tool

This kind of tool can potentially make the vocabulary ontologies more interesting, concrete, and easier to comprehend. Changes in context can be seen in a more illustrative manner than observing changes in raw OWL files. Application developers can see this operation as in real life and notice practical errors better without expensive laboratory's tests.

The tool should also offer some querying possibilities over the semantic models, and enable the users or service developers to define their own simple RDQL or SPARQL queries via a convenient graphical user interface.

A proof of concept prototype of the tool has been implemented with Java Swing for graphical UI and Jena OWL API for managing and accessing the ontologies (Figure 3-3).

3.4 Changes to vocabularies from previous iteration

This chapter describes the changes/modifications performed on vocabularies that support semantic description of Amigo services issued for the present document with respect to the ones released in the previous Deliverable D3.2 [Amigo-D3.2].

3.4.1 Changes supporting visualization of vocabularies

The visualization tool considers only individuals of the *VisualComponent* class ignoring all data that is irrelevant or impossible to visualize. *VisualComponent* class has two subclasses: *Item* and *Area*.

These concepts used by the visualization tool can be linked, for example, to the Context ontology concepts by subclassing *Context:Object* and *Context:Person* from the Item class, and *Context:Area* from the Area Class. Using this approach, the visualization is independent of any changes to Amigo vocabulary ontologies, and provides the developer with some control on what kind of things should be visualized.

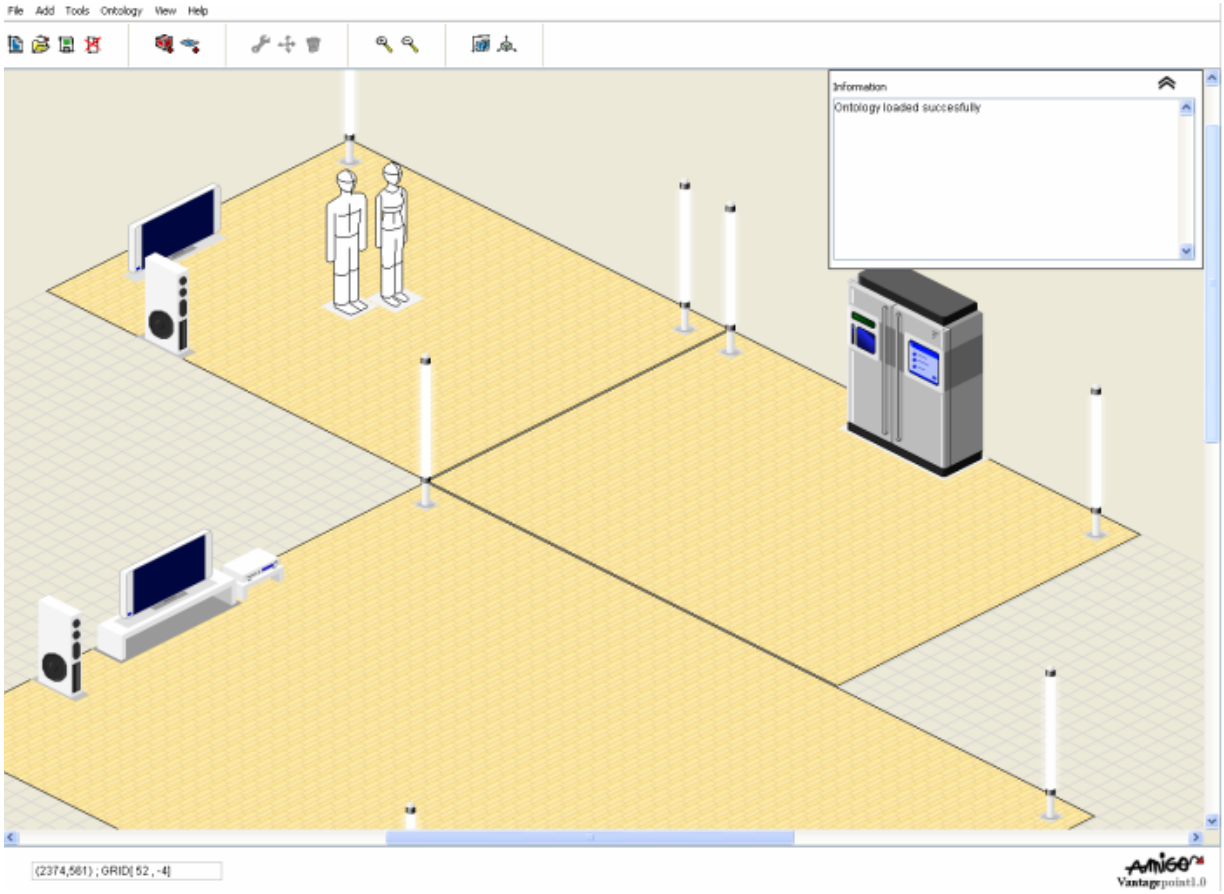

Figure 3-3: A screen capture of the Amigo vocabulary visualization tool

3.4.2 Multimedia content vocabularies

The Multimedia content vocabulary has been upgraded by the introduction of several properties (see Figure 3-4). For example, the *hasResource* property has been introduced, relating individuals of the *MultimediaContent* class (representing the abstract concept of content) to individuals from the *MultimediaResource* class (representing binary files). Furthermore, data type properties that relate individuals from *MultimediaContent* to different kinds of information semantically modeling properties of content such as *Title, Date*, etc. have been included in the model, thus providing a basic description of Content that may, due to intrinsic extensibility of ontologies, be further completed and refined providing new properties with specialized ranges. Several properties define low-level features of resources such as *hasBitRate* or *hasFormat* that are essential for the Amigo Content Adaptation subcomponent of the Amigo middleware (see Section 8.3).

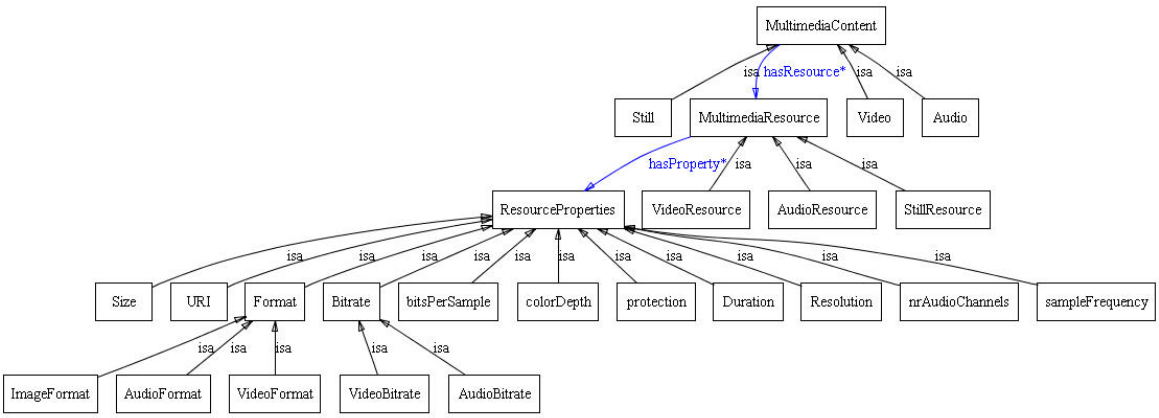

Figure 3-4: MultimediaContent *Ontology*

The *MultimediaContent* ontology enables semantic description of transcoding services, providing the basic vocabulary for describing IOPEs. The Amigo Content Adaptation subcomponent of the middleware uses semantic descriptions of transcoding plug-ins, in order to provide for a given initial resource with given features a wider range of possible output resources, through composition, than by considering plug-ins separately. In this component, the semantic description of plug-ins is based on OWL-S and SWRL, and uses Multimedia Content ontologies to model IOPEs and auxiliary XML Schema data types. An example of a possible plug-in that can transform an audio resource with AAC format into one with MP3 format with no restrictions in other properties is given in Section 3.5.

Furthermore, a mapping ontology has been produced (see Figure 3-5), in order to provide an extensible framework for translating – with some restrictions explained below and semi-transparently – common properties of content expressed in any XML-based language into a consistent *MultimediaContent* individual (see Figure 3-6). This enables translation to be a transparent process, although some preprocessing of the source XML-based description is required: the original document must be transformed into a key-value map.

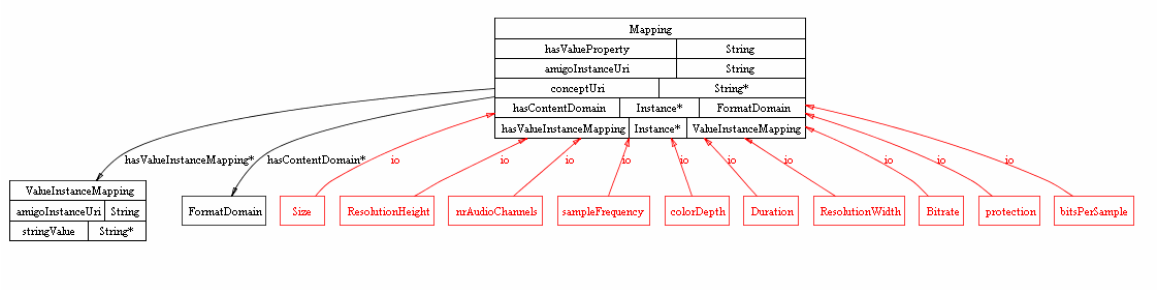

Figure 3-5: The Mapping *ontology*

The immediate restriction derived from the key-value map is that one single element defining a given feature of content is translated. Therefore, multiple elements defining different values for the same feature are not considered. One exception to this is *resources*, which are processed separately in order to support multiple *resources* for the same content. Another restriction is the data type of the values: this must be processed by the translation engine and, therefore, an agreement between translation engine and preprocessor must be achieved. A reference implementation can be found in the Content Description Interoperability module included in the

Content Adaptation subcomponent: this implementation uses: strings as base types for communication with the preprocessor; and some enumerated derived types for some content or resource features (e.g., MIME-type string for “format” concept), which must be explicitly included in the ontology mapping when referring to ontology instances, rather than data type values.

Concepts available for mapping are instances of the Mapping class, and have several properties that enable translation and several ones that optimize it. For example, *amigoConceptUri* is a data-type property specifying the concept in the Amigo *MultimediaContent* ontology. This URI may represent a Class or a Property. The *conceptUri* relates this concept with other concepts in the form of URIs. These URIs should be chosen so that minimum preprocessing of the original XML document is required (i.e., some variation of fully qualified element names), since this should be used as keys in the maps passed to the translation engine. The *hasContentDomain* property enables pre-selection of mappings, once the translation engine is able to determine the type of the content (i.e., Audio, Image, Video, etc.) related to this metadata. The two other properties with *Mapping* as domain describe how the value taken by this concept should be inserted in the resulting instance of the Amigo *MultimediaContent* ontology:

- The property *hasValueProperty* defines the property of the new instance of the class defined by *amigoConceptUri* that relates it to the value it takes.
- The property *hasValueInstanceMapping* relates this concept mapping to an instance of the *ValueInstanceMapping*: this class represents mappings of certain enumerated possible values of a concept to an instance in the *MultimediaContent* ontology. This is the case of formats: formats are represented as instances.

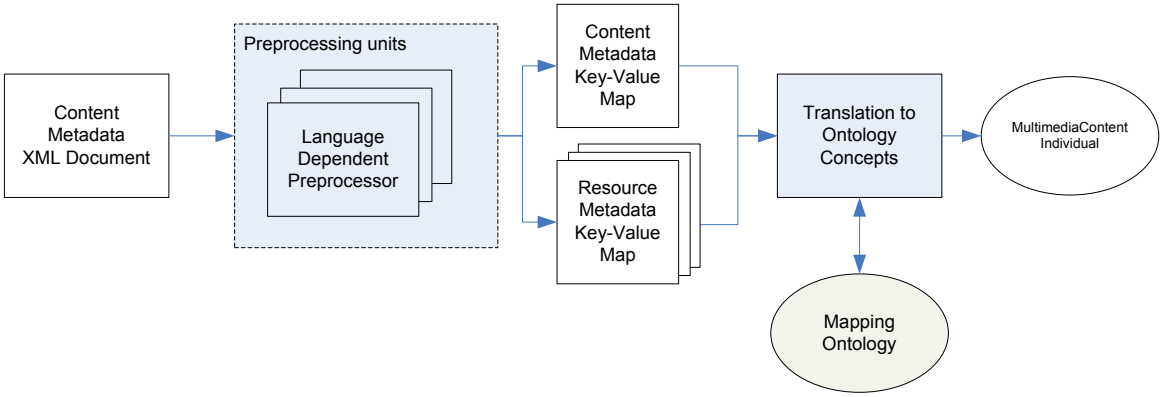

Figure 3-6: Architecture for translation to Amigo MultimediaContent Ontology concepts

3.4.3 Structural Changes to Context and QoS ontologies

The changes related to Context and QoS Ontologies refer to the fact that the Language ontology has been interrelated with the Vocabulary ontology, while in the previous version the former was independent and unaware of the latter.

More specifically, the *ContextLanguage* ontology includes all the information that is required for the instantiation of the various different vocabulary subclasses, and now imports the *ContextVocabulary* ontology. Thus, a hierarchical structure has been established where the *ContextParameter* (of the ContextLanguage ontology) is the super-class of all the *ContextConcept* classes (of the ContextVocabulary ontology). These ContextConcept classes are the root classes of the ContextVocabulary. Thus, the situation is as follows: the

UserContext class, for example, is defined in the ContextVocabulary and is a subclass of the *ContextConcept*, which in turn is a subclass of the *ContextParameter* that is the root class of the ContextLanguage ontology. The necessary changes have been performed to both context ontologies to guarantee the consistency of the implemented approach. As a consequence, the ContextVocabulary ontology now imports, in addition to the Amigo core ontology, the ContextLanguage ontology as well, while the ContextLanguage now imports the ContextVocabulary ontology.

Similar changes have been performed to the *QoSVocabulary* and *QoSLanguage* ontologies.

3.4.4 Non-structural Changes to Context and QoS ontologies

In this subsection, the slight modifications applied to some classes of the aforementioned ontologies are described. With regard to the classes of the User Context Domain Vocabulary Ontology, some information has been added in the form of new object properties, datatype properties and subclasses, in order to fulfill the requirements of the UMPS (Task 4.2: User Modelling and Profiling Service). More specifically, apart from some object properties that have been added to the *User* class, several datatype properties have been added to the *PersonalDetails* class, such as the eyeColor, the imgPrint, etc. Additionally, several subclasses of the *Preferences* class have been added to address the requirements of the UMPS.

In Figure 3-7, the User and PersonalDetails classes are depicted, as designed by the OntoViz plug-in of the Protégé ontology editor. Finally, in Figure 3-8, the subclasses of the Preferences class are illustrated, as designed by OntoViz.

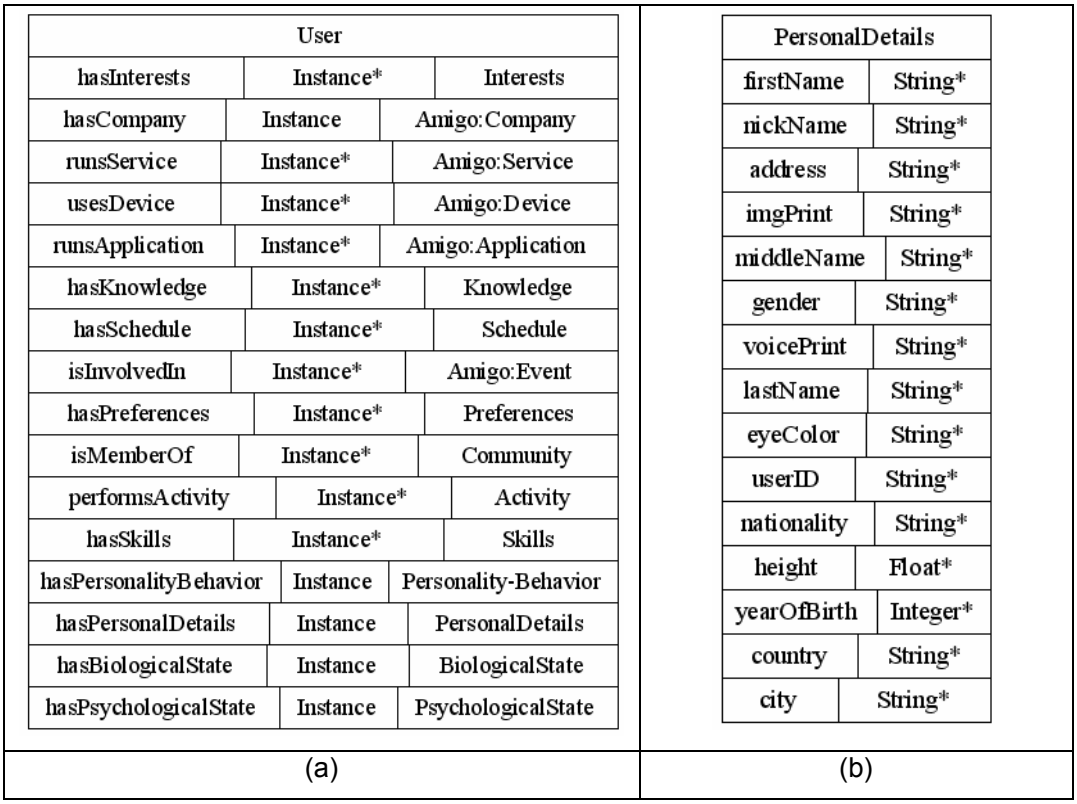

Figure 3-7: The updated User and PersonalDetails classes of the Context Vocabulary Ontology

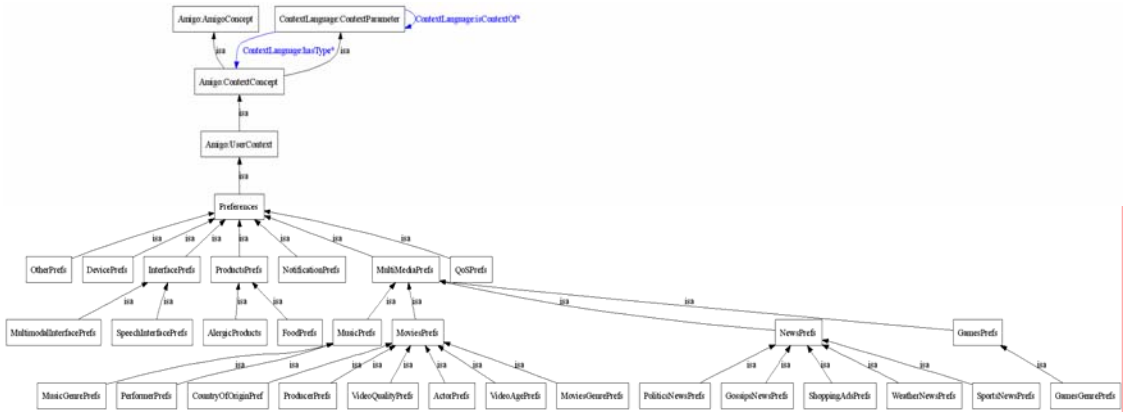

Figure 3-8: The subclasses of the Preferences class of the Context Vocabulary Ontology

3.5 Examples

As an example of the use of multimedia ontologies, the following is a possible plug-in that is able to transform an audio resource with AAC format into one with MP3 format with no restrictions in other properties:

```
<?xml version="1.0"?>
<!DOCTYPE owl [
      <!ENTITY xsd "http://www.w3.org/2001/XMLSchema" >
      <!ENTITY multimedia "http://www.owl-ontologies.com/Amigo/Multimedia.owl" >
]>
<rdf:RDF
   xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
   xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
   xmlns:swrl="http://www.w3.org/2003/11/swrl#"
   xmlns:time="http://www.isi.edu/~pan/damltime/time-entry.owl#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:expr="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
   xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
   xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:j.0="http://protege.stanford.edu/plugins/owl/protege#"
   xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:amigomultimedia="http://www.owl-ontologies.com/Amigo/Multimedia.owl#"
   xmlns:amigoserviceregistry="http://www.owl-ontologies.com/Amigo/ServiceRegistry.owl#"
   xmlns="http://www.owl-ontologies.com/Amigo/serviceComp1.owl#"
   xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://www.owl-ontologies.com/Amigo/serviceComp1.owl">


  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.daml.org/rules/proposal/swrlb.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/rules/proposal/swrl.owl"/>
    <owl:imports rdf:resource="http://www.owl-ontologies.com/Amigo/Multimedia.owl"/>
  </owl:Ontology>
```

```
<owl:Class rdf:ID = "AudioResourceAAC">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="http://www.owl-ontologies.com/Amigo/Multimedia.owl#AudioResource"/>
    <owl:Restriction>
       <owl:onProperty rdf:resource="http://www.owl-ontologies.com/Amigo/Multimedia.owl#hasProperty"/>
       <owl:hasValue rdf:resource="http://www.owl-ontologies.com/Amigo/Multimedia.owl#AAC"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID = "AudioResourceMP3">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="http://www.owl-ontologies.com/Amigo/Multimedia.owl#AudioResource"/>
    <owl:Restriction>
       <owl:onProperty rdf:resource="http://www.owl-ontologies.com/Amigo/Multimedia.owl#hasProperty"/>
       <owl:hasValue rdf:resource="http://www.owl-ontologies.com/Amigo/Multimedia.owl#MP3"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>


<service:Service rdf:ID="serviceComp1">
<service:presents>
<profile:Profile rdf:ID="AACtoMP3FormatConverterProfile">
   <service:presentedBy rdf:resource="#serviceComp1"/>

   <profile:hasInput>
    <process:Input rdf:ID="AudioResourceInput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
      >http://www.owl-ontologies.com/Amigo/serviceComp1.owl#AudioResourceAAC</process:parameterType>
    </process:Input>
  </profile:hasInput>

  <profile:hasInput>
    <process:Input rdf:ID="TargetAudioFormatInput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
      >http://www.owl-ontologies.com/Amigo/Multimedia.owl#AudioFormat</process:parameterType>
    </process:Input>
  </profile:hasInput>

  <profile:hasOutput>
    <process:Output rdf:ID="AudioResourceOutput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
      >http://www.owl-ontologies.com/Amigo/Multimedia.owl#AudioResource</process:parameterType>
    </process:Output>
  </profile:hasOutput>


        <profile:hasResult>
      <process:Result>
      <process:hasEffect>
              <expr:SWRL-Condition rdf:ID="MP3Effect">
          <expr:expressionBody rdf:parseType="Literal">
              <swrl:AtomList xmlns:swrl="http://www.w3.org/2003/11/swrl#">
              <rdf:first xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
                                  <swrl:ClassAtom>
```

```
                                                    <swrl:classPredicate
rdf:resource="#AudioResourceMP3"></swrl:classPredicate>
                                                    <swrl:argument1 rdf:resource="#AudioResourceOutput"></swrl:argument1>
                                                    </swrl:ClassAtom>
                        </rdf:first>
                                <rdf:rest xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"></rdf:rest>
                                </swrl:AtomList>
                    </expr:expressionBody>
                    <expr:expressionLanguage rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#SWRL"/>
                    </expr:SWRL-Condition>
                    </process:hasEffect>
                </process:Result>
                </profile:hasResult>
        </profile:Profile>
    </service:presents>
  </service:Service>
</rdf:RDF>
```

Three examples for using the Context and QoS ontologies are provided in the following. The first example concerns the representation of a user, which instantiates the User class of the ContextVocabulary ontology. This first example is based on the current implementation of the ontologies and is presented below.

```
<User>
        <identifier>amigo_user783@Amigo</identifier>
        <hasPersonalDetails>
                <PersonalDetails>
                        <identifier>personal_details#347</identifier>
                </PersonalDetails>
        </hasPersonalDetails>
        <hasPreferences>
                <speechInterfacePrefs>
                        <identifier>speechIFprefs#865</identifier>
                <speechInterfacePrefs>
                <DevicePrefs>
                        <identifier>devicePrefs#341</identifier>
                </DevicePrefs>
        </hasPreferences>
        <performsActivity>
                <Activity>
                        <identifier>activity#873</identifier>
                </Activity>
        </performsActivity>
        <usesDevice>
                <Device>
                        <identifier>FSN560</identifier>
                </Device>
        </usesDevice>
        <isLocatedIn>
                <Building>
                        <identifier>NTUA_Building#112</identifier>
                </Building>
        </isLocatedIn>
```

```
</User>
```

However, following the approach above does not allow us to attach to the object properties of the ContextConcepts metadata information such as timestamp, probability or accuracy. This requirement has been identified recently by Task 4.1 (Context Management Service). We are currently working on producing an enhanced version of the context ontologies that addresses this requirement. The solution proposed consists in replacing all object properties of the ContextVocabulary classes (i.e., direct associations between ContextConcepts) by new ContextParameter subclasses carrying object properties with the classes previously associated directly. These new ContextParameter subclasses of the ContextLanguage ontology bind indirectly the existing ContextConcept subclasses of the ContextLanguage ontology and can be enriched with the necessary metadata information. As the necessary changes on the context ontologies are not finished yet, they are not mentioned in the previous subsection. However, we believe that an example of usage of this future version of the ontologies is valuable at this stage of the project and should thus be included in the present document. Such an example follows and provides the representation of the same user as before, but this time based on the future implementation of the context ontologies. In this example, the *UserDevice*, *UserLocation* and *UserActivity* classes have replaced the object properties between the *User* class and the *Device*, *Place* and *Activity* classes respectively.

```
<User>
        <identifier>amigo_user783@Amigo</identifier>
        <hasPersonalDetails>
                <PersonalDetails>
                        <identifier>personal_details#347</identifier>
                </PersonalDetails>
        </hasPersonalDetails>
        <hasPreferences>
                <speechInterfacePrefs>
                        <identifier>speechIFprefs#865</identifier>
                <speechInterfacePrefs>
                <DevicePrefs>
                        <identifier>devicePrefs#341</identifier>
                </DevicePrefs>
        </hasPreferences>
        <performsActivity>
                <UserActivity>
                        <identifier>userActivity#543</identifier>
                </UserActivity>
        </performsActivity>
        <usesDevice>
                <UserDevice>
                        <identifier>userDevice#901</identifier>
                </UserDevice>
        </usesDevice>
        <isLocatedIn>
                <UserLocation>
                        <identifier>userLocation#161</identifier>
                </UserLocation>
        </isLocatedIn>
</User>

<UserDevice>
```

```
        <identifier>userDevice#901</identifier>
        <timestamp>01-09-2006T13:39:20CET</timestamp>
        <probability>0.87</probability>
        <UD_User>
        <User>
                        <identifier>amigo_user783@Amigo</identifier>
                </User>
        </UD_User>
        <UD_Device>
        <Device>
                <identifier>FSN560</identifier>
        </Device>
        </UD_Device>
</UserDevice>

<UserActivity>
        <identifier>userActivity#562</identifier>
        <timestamp>01-09-2006T13:39:20CET</timestamp>
        <probability>0.64</probability>
        <UA_User>
        <User>
                        <identifier>amigo_user783@Amigo</identifier>
                </User>
        </UA_User>
        <UD_Activity>
        <Activity>
                <identifier>activity#873</identifier>
        </Activity>
        </UD_Activity>
</UserActivity>

<UserLocation>
        <identifier>userLocation#754</identifier>
        <timestamp>01-09-2006T13:39:20CET</timestamp>
        <probability>0.73</probability>
        <Accuracy>5</accuracy>
        <UL_User>
        <User>
                        <identifier>amigo_user783@Amigo</identifier>
                </User>
        </UL_User>
        <UD_Place>
        <Building>
                <identifier>NTUA_Building#112</identifier>
        </Building>
        </UD_Place>
</UserLocation>
```

Finally, the third example presented in this section concerns the representation of a service that has a specific QoS profile; it is instantiated from the Service class and is depicted below:

```
<Service>
        <identifier>Service#895</identifier>
        <deployedOn>
```

```
            <Device>
                <identifier>FSN560</identifier>
            </Device>
        </deployedOn>
        <hasFunctionalCapability>
            <FunctionalCapability>ExtendedHomeSupport
            </FunctionalCapability>
        </hasFunctionalCapability>
        <hasQoSParameter>
            <QoSConcept>
                <MTBF>1000</MTBF>
            </QoSConcept>
            <QoSConcept>
                <ErrorRate>0.005</ErrorRate>
            </QoSConcept>
            <QoSConcept>
                <ResponseTime>0.02</ResponseTime>
            </QoSConcept>
            <QoSConcept>
                <Availabilty>0.95</Availability>
            </QoSConcept>
            <QoSConcept>
                <Accessibility>0.90</Accessibility>
            </QoSConcept>
            <QoSConcept>
                <Accuracy>0.99</Accuracy>
            </QoSConcept>
            <QoSConcept>
                <Security>SSL</Security>
            </QoSConcept>
            <QoSConcept>
                <Cost>19.95</Cost>
            </QoSConcept>
        </hasQoSParameter>
</Service>
```

4 Service description – Service discovery, composition, adaptation & execution

4.1 Introduction

The main objective of this chapter is to introduce a comprehensive approach to service description, discovery, composition, adaptation and execution in the Amigo environment. These aspects are very much interrelated and lead us to introduce a unified architecture called SD-SDCAE (Service Description – Service Discovery, Composition, Adaptation & Execution). Service description is done using the Amigo-S language [Amigo-D3.2] and the Amigo vocabulary ontologies (see Chapter 3). Amigo-S enables to semantically describe the functionalities offered by services, regardless of their underlying deployment and execution framework. Nevertheless, in our approach to SD-SDCAE, we assume that possible middleware-layer heterogeneity can be and has been tackled by applying solutions supported by the Amigo programming & deployment framework (see Chapter 2) or based on the middleware interoperability mechanisms (see Chapter 5); thus, we design SD-SDCAE on top of a homogeneous middleware; for practical reasons we employ Web Services, even if our solution is generic enough to be independent of the underlying service infrastructure. In the following sections, we first introduce the high-level architecture of SD-SDCAE (Section 4.2). Then, we elaborate on two specific aspects of SD-SDCAE, namely, service discovery (Section 4.3) and service composition (Section 4.4).

4.2 High-level architecture of SD-SDCAE

User applications use services deployed in the Amigo home. In the static case, we know in advance which single or multiple services we need to invoke or to compose. We also know the interfaces and the behavior of these services. These services may be looked up by name and invoked employing the basic service discovery and service interaction APIs (see Chapter 2). However, in the dynamic case, we do not know in advance which services to employ nor their exact interfaces and behaviors. We thus rely on discovery of services based on the semantics of required functionalities. For this, both our “approximate” request (since we do not know in advance the services that we will finally employ) and the available provided services should be semantically described. Then, we should carry out: semantic service discovery; service composition if no single service satisfies our request but the composite usage of several services does; and adaptation of our “approximate expectation” to the available service(s). We finally execute the “adapted expectation” invoking the single or multiple composed services. Some intermediate (less dynamic) cases may also be identified between the two above extremes.

The general functional architecture of service discovery, composition and adaptation in the Amigo system is illustrated in Figure 4-1. Devices in the Amigo environment may undertake three different roles, which communicate through the common SDCAE interface. The first role is the “user device”, which executes an application, but needs to access services on remote devices to obtain certain functionalities. The application should describe semantically the functionalities that it requires. This description is called ‘abstract’ because it does not refer to any existing service; moreover, the ‘concretization’ needed for the application to actually invoke a service offering a semantically adequate functionality can be different from one service to another. The abstract description should then be refined – based on the discovered services’ interfaces – to a concrete description, which can be executable within the actual environment. The second role is the service repository, which stores all deployed services in a given environment. It offers an interface for registering and un-registering services, and an interface for searching for services. The third role is the one of the service provider, who is

responsible for registering its services with the repository. Interactions between the three roles and their related internal actions are indicated in Figure 4-1 as a sequence of functional steps (1-9). These steps are briefly discussed in this section, while a subset of them is elaborated in the rest of the sections of this chapter.

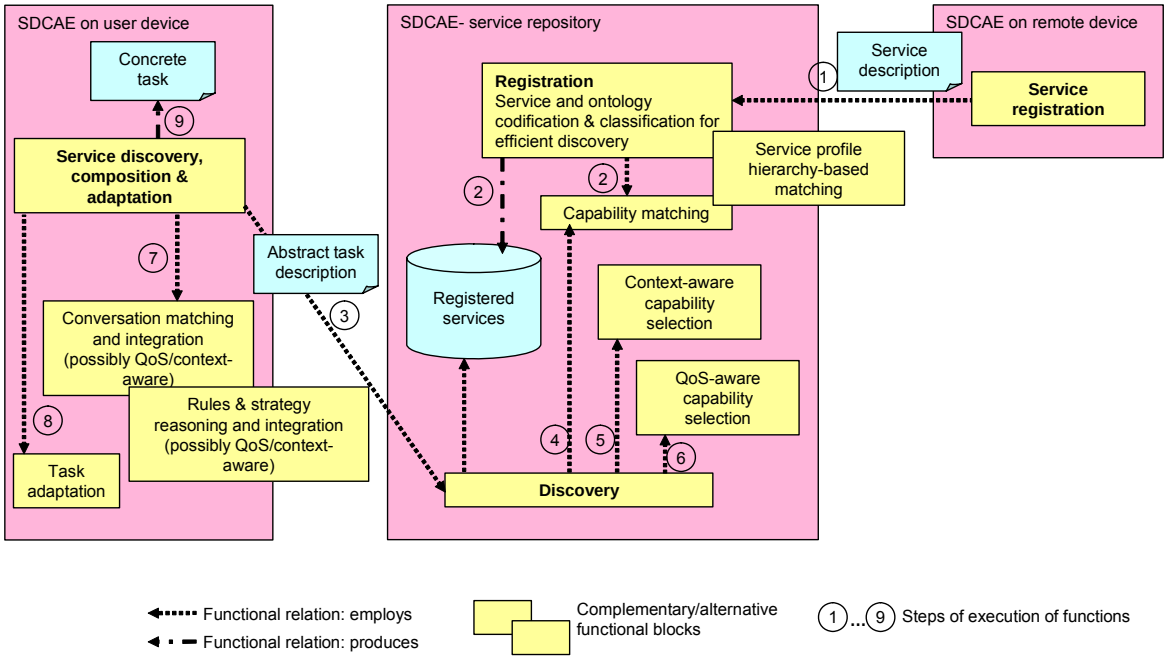

Figure 4-1: Service discovery, composition & adaptation - Functional architecture

A user application on a user device comprises a number of parts (see Figure 4-2):

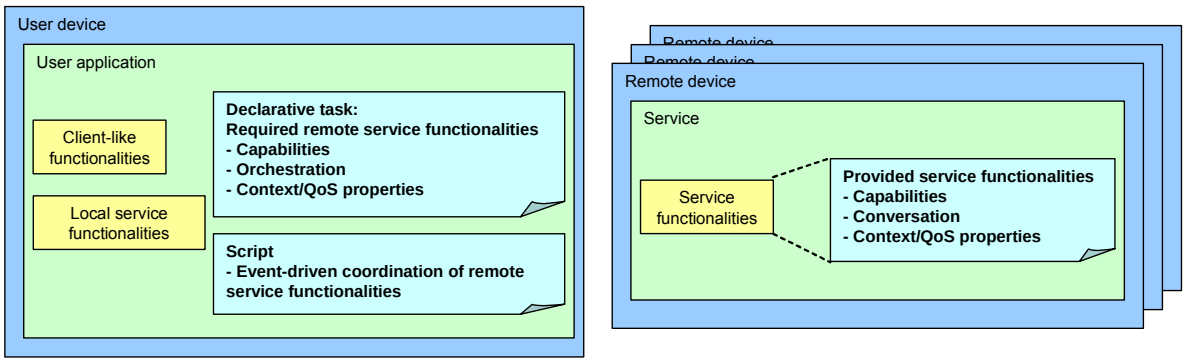

Figure 4-2: Application/service description & coding

Task will compose, orchestrate and adapt to remote service functionalities. It employs Amigo-S to describe a workflow of activities corresponding to the usage of several functionalities that should be sought on remote available services. Functionalities are described abstractly in Amigo-S and by using vocabulary ontologies, and should be refined to concrete services based on the discovered available services that can provide the functionalities.

Client-like functionalities may do initialization and "glue" all the other parts. It is a hard-coded part that is inflexible. It initiates semantic discovery, adapts the task to make it executable, and initiates the execution of the workflow.

Script will compose remote service functionalities in an event-driven way. Events and associated functionalities that should be executed are described in a scripting language.

Local service functionalities may conveniently be used by either task or script. They correspond to local computation done for internal processing of data coming from remote services.

The service discovery protocol is illustrated in Figure 4-3. The service repository is (or seen as) a centralized entity that is accessed by service providers to register their services and by service requestors to find deployed services. The registry itself can be found by both service providers and requestors using a legacy service discovery protocol. Service providers register their deployed services with the registry by sending the Amigo-S semantic description of the service (or a URI reference to it) (Steps 1, 2 of Figure 4-1). A user application can then issue discovery requests to the service repository to find a service based on a semantic description. The query can be for a single functionality, or for a full task description consisting of several functionalities described in a workflow (Step 3). In all cases, we consider that the query is an Amigo-S document, where capabilities are "required capabilities". The repository will then match the query to the descriptions of the registered services (Step 4), and return the Amigo-S documents of one or more discovered fitting services, or URI references to them, which can then be used to invoke the services directly.

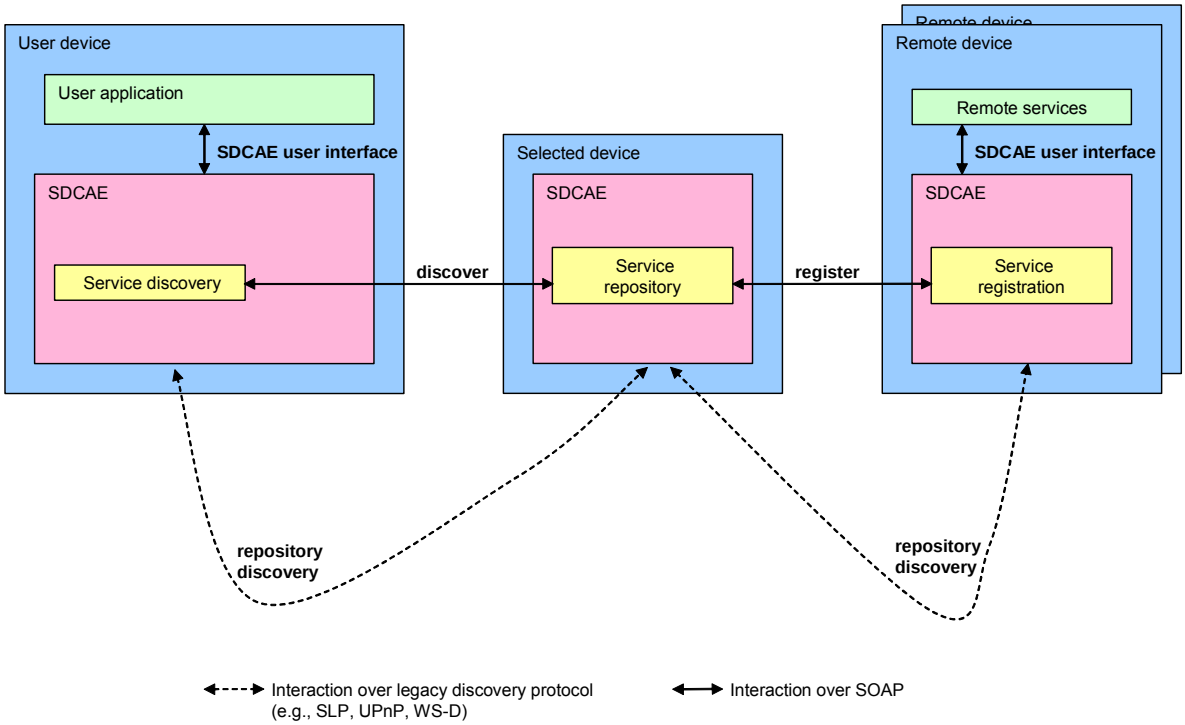

Figure 4-3: Service discovery protocol

If the whole remote functionality required by the abstract task is offered by a single service from the discovered service set, then the abstract task is adapted to the concrete features of this service to produce a concrete task. If no single service is sufficient, service composition is carried out by conversation matching and integration based on the abstract task description

(Step 7). Again then, the abstract task is adapted to the concrete features of the composed services to produce a concrete task (Steps 8, 9). Context- and QoS-related properties are used to refine search results (on the repository) for obtaining services that satisfy the respective properties of the application (Steps 5, 6).

Once the abstract task description is refined to a concrete task, it can be executed: The initialization is done by the hard-coded part, which contains calls to an orchestration execution engine; the latter executes the produced concrete task and orchestrates the composed remote services and the local service functionalities (see Figure 4-4). We assume further that the event-driven script part of the user application may be separately executed by a script execution engine, and possibly compose as well remote and local service functionalities. The combination of the orchestration-driven and the event-driven parts of the application is an issue that is still to be investigated.

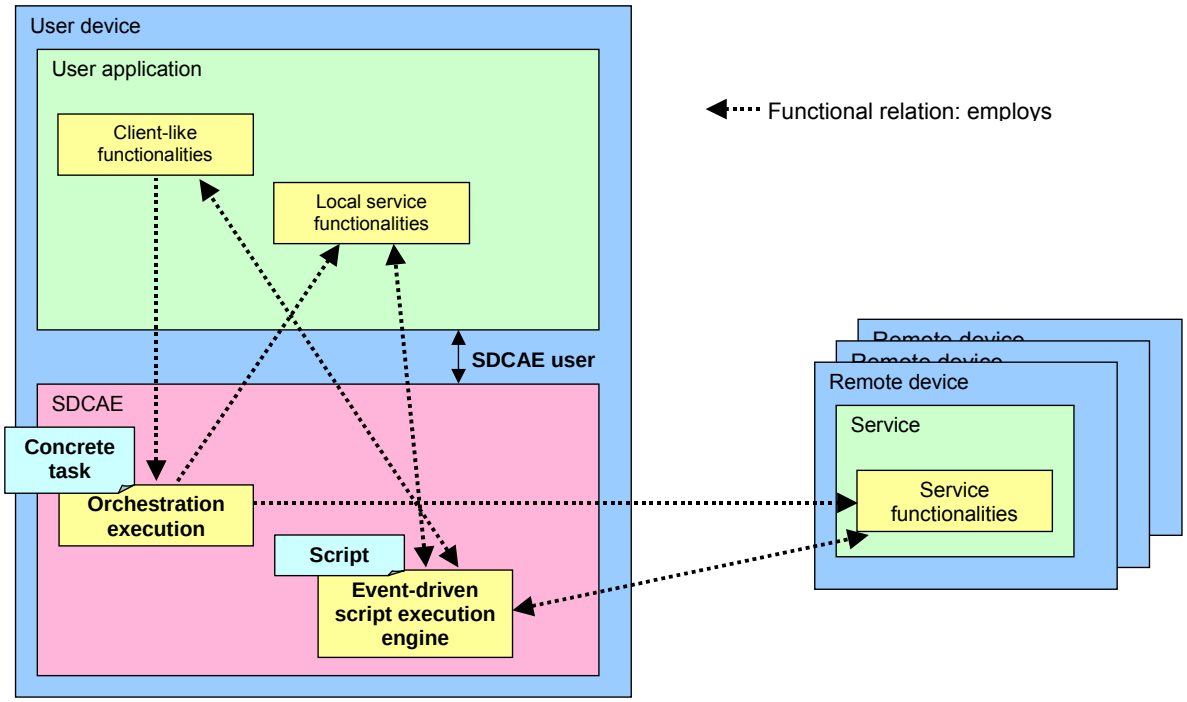

Figure 4-4: Service execution – Functional architecture

4.3 Service registration and discovery

Discovery of available services that provide a given capability is done by semantic matching (Step 4 of Figure 4-1) in the service repository between the services' capabilities that have been registered (Step 2) and the capabilities that are sought. In the following two sections, we propose two – partly complementary, partly alternative – approaches to efficient semantic service matching based on a number of optimizations performed upon the registration of services in the service repository. Specifically, we discuss capability-based matching (Section 4.3.1) and service profile hierarchy-based matching (Section 4.3.2). In these approaches, the inputs, outputs and the category of the required capabilities are matched against the ones of semantically equivalent capabilities offered by deployed services. Combining these approaches to semantic service matching within SD-SDCAE is still an open issue. Following the semantic service matching which leads to a discovered service set, this set is refined by context-aware (Step 5) and QoS-aware (Step 6) service selection. We discuss these issues in

Sections 4.3.3 and 4.3.4, respectively. We have to note that at the current stage, the context-aware and QoS-aware service selection are elaborated on the basis of their proper fundamental usage scenarios; their integration into SD-SDCAE is still an open issue and is considered as future work.

4.3.1 Efficient semantic service matching

The semantic service discovery protocol should offer performance that makes it appropriate for use in highly dynamic networked environments populated by resource-constrained, wireless devices. This is a major challenge due to the poor performance and resource costs of ontology-based semantic reasoning. This has led us to introduce a solution to lightweight semantic matching of Web services towards the actual exploitation of semantic Web services in AmI environments, and more particularly in the Amigo home environment. Our solution optimizes ontology-based semantic reasoning, which is at the heart of the matching process. Furthermore, we propose a classification of service advertisements within the service repository towards efficient access and retrieval of services.

An example of service profiles as enabled by Amigo-S (restricted to service inputs, outputs and category) is depicted in the upper part of Figure 4-5. Along with service descriptions, the figure includes in its lower part two ontologies representing the concepts employed in the service descriptions. The service on the PDA requires a capability named GetVideoStream, which belongs to the service category VideoServer, takes as input a title of a VideoResource and provides as output an actual Stream. The service on the workstation provides two capabilities, SendDigitalStream and ProvideGame, which share common attributes such as the workstation resources available to them. For the former, service category is DigitalServer, input is DigitalResource, and output is Stream, while for the latter, service category is GameServer, input is GameResource and output is Stream. These two capabilities are dependent, as SendDigitalStream includes ProvideGame, but are separately accessible. Thus, a peer service (in other words, a client) may access the former and have the option to access a video resource, a sound resource or a game; or access the latter, asking specifically for a game. The peer service on the PDA asking for a video resource should access SendDigitalStream, which also includes GetVideoStream. Making the right choice is supported by service matching, which is described in the following sections.

Semantic Matching Relation

Based on the Amigo-S service specification, we define a matching relation, i.e., Match(C_1, C_2), which allows identifying whether capability C_1 subsumes (is equivalent or includes) capability C_2 , i.e., if C_1 can substitute C_2 in the provisioning of a service capability that is semantically characterized by C_2 (see the example of SendDigitalStream and GetVideoStream in Figure 4-5). The Match relation then constitutes the basis of service discovery, as seeking a capability characterized by C amounts to discovering any networked service advertising a capability described by N such that Match(N, C) holds. Additionally, the Match relation may conveniently be exploited to group similar capabilities of networked services towards efficient service discovery, as further presented in the next section.

Specifically, the Match relation is defined using the function distance($concept_1$, $concept_2$), hereafter denoted by d($concept_1$, $concept_2$), which gives the semantic distance between two concepts $concept_1$ and $concept_2$ in the classified[11] ontology to which the concepts belong. Precisely, if $concept_1$ does not subsume $concept_2$ in the ontology to which they belong, the distance between the two concepts does not have a numeric value, i.e., d($concept_1$,

11 Ontology classification is the result of semantic reasoning on ontology specifications. It allows inferring implicit relationships between concepts from the explicit definitions of these concepts.

concept$_2$)=NULL. Otherwise, i.e., if concept$_1$ subsumes concept$_2$, the distance takes as value the number of levels that separate concept$_1$ from concept$_2$ in the ontology hierarchy.

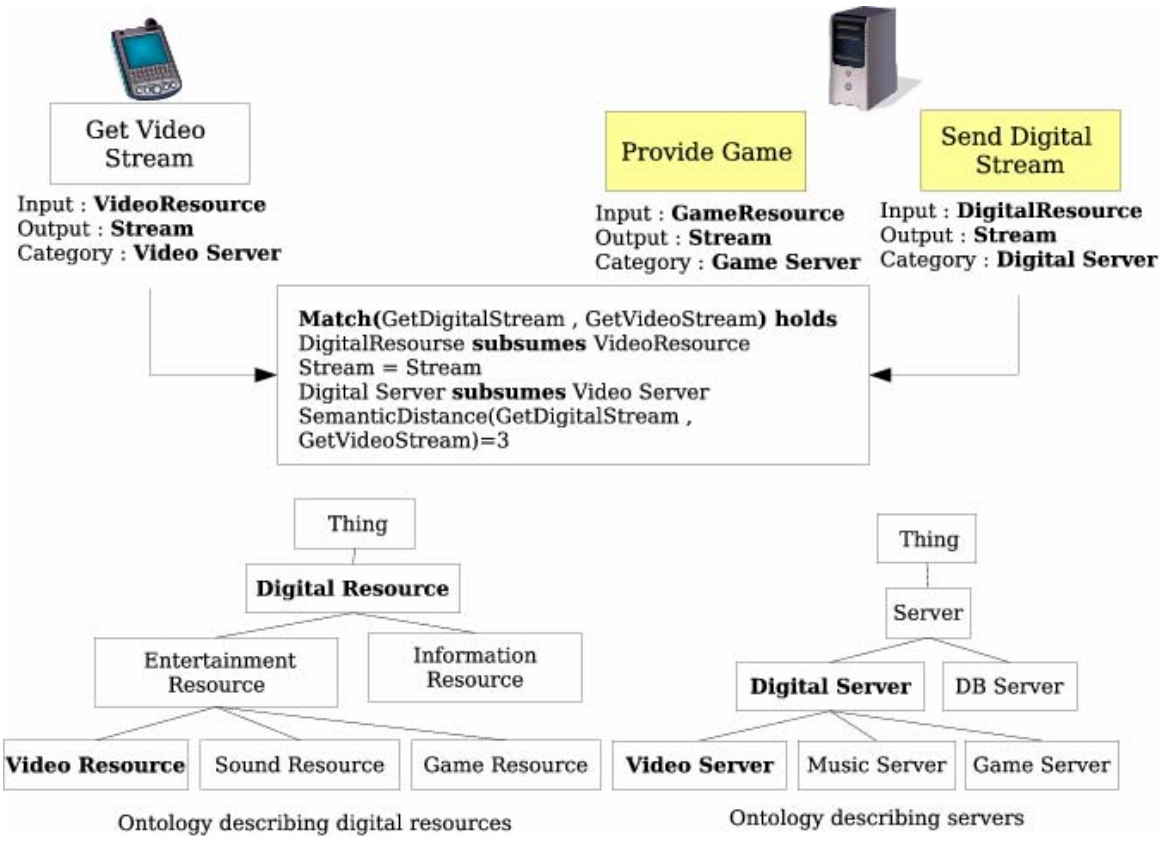

Figure 4-5 Describing and matching capabilities of services

Formally, let the provided capability C_1 be defined by the set of expected inputs C_1.In and set of offered outputs C_1.Out, and the required capability C_2 be defined by the set of offered inputs C_2.In and the set of expected outputs C_2.Out. Furthermore, let the capability C_1 define a set of provided properties C_1.P, and the capability C_2 define a set of required properties C_2.P, where these properties describe all the information that can be required in the user request such as the service category and non-functional properties; currently, we only consider the former property. The relation Match is then defined as:

$$\begin{aligned} Match(C_1, C_2) = & \forall in' \in C_1.In, \exists in \in C_2.In : d(in, in') \geq 0 \text{ and} \\ & \forall out' \in C_2.Out, \exists out \in C_1.Out : d(out, out') \geq 0 \text{ and} \\ & \forall p' \in C_2.P, \exists p \in C_1.P : d(p, p') \geq 0 \end{aligned}$$

From the above, the relation Match(C_1, C_2) holds if and only if all the expected inputs of C_1 are matched with inputs offered by C_2, all the expected outputs of C_2 are matched with outputs offered by C_1, and all the required properties of C_2 are matched with properties provided by C_1.

Furthermore, we define the function SemanticDistance(C_1, C_2), which gives the semantic distance between the capability C_1 and the capability C_2:

$$SemanticDistance(C_1, C_2) = \sum_{i=1}^{n_1} d(C_2.In_i, C_1.In_i) + \sum_{i=1}^{n_2} d(C_1.Out_i, C_2.Out_i) + \sum_{i=1}^{n_3} d(C_1.p_i, C_2.p_i)$$

where n_1 is the number of inputs expected by C_1, n_2 is the number of outputs expected by C_2, and n_3 is the number of additional properties required by C_2. The semantic distance between capabilities corresponds to the sum of the distances between the pairs of related concepts in the advertisement and the request. This allows scoring service advertisements with respect to the requested capability with which they are being compared, and selecting the advertisement whose description best fits the user's requirements. An example of matching semantic service capabilities is shown in the middle part of Figure 4-5. In the figure, the requested capability GetVideoStream is matched with the provided capability SendDigitalStream, by using the two underlying ontologies describing digital resources and servers. The relation Match(SendDigitalStream, GetVideoStream) holds, and the semantic distance between these capabilities is equal to 3.

Implementation and evaluation of semantic matching of service capabilities using existing OWL reasoners has been presented in the Amigo Deliverable D3.2 [Amigo-D3.2]. Related work has also been carried out in [BKGI06]. Results show that matching semantic service capabilities is a computation-intensive task with high response times compared to classic syntactic-based service discovery protocols. In particular, the most expensive phase is that of semantic reasoning. Such poor performance and resource consumption is not acceptable for a service discovery protocol aimed at AmI environments, where service discovery needs to be efficient enough to ensure service availability despite the network's dynamics, and lightweight enough for use by thin, wireless devices. Thus, in order to enable actual deployment of semantic Web services in AmI environments, a number of optimizations have to be introduced in the process of matching semantic service capabilities, particularly targeting acceptable response times. The next sections introduce such solutions, building upon recent efforts in the area of efficient semantic service matching.

Achieving lightweight discovery of semantic Web services

Lightweight discovery of semantic Web services requires minimizing the overhead due to semantic reasoning, possibly performing it off-line so that semantic reasoners do not need to be used when advertising and seeking networked services; moreover, we attempt to minimize the overhead due to service matching. Specifically, optimization can be introduced at two levels. First, at the semantic reasoning level, by reducing the time spent to infer relationships between concepts in ontologies. Second, at the service discovery level, by classifying services within the service repository in a way that reduces the number of semantic matches performed to answer a user request. Related optimizations for both ontology-based semantic reasoning and classification of service advertisements have been proposed in the literature [CF03, SPS04].

Using the matching relation defined in the previous section, we propose an efficient semantic service discovery protocol for AmI environments. Efficiency is addressed in terms of response time for both the discovery and advertisement of service capabilities. Towards this goal, we present below a number of optimizations of the semantic matching process. First, in order to reduce the time to load and classify ontologies, which is the most costly phase in the discovery process, we propose to encode classified ontologies. Then, in order to reduce the number of semantic matches performed in the querying phase, we propose to classify capabilities of networked services into hierarchies.

Encoding Concept Hierarchies

In order to avoid performing semantic reasoning at runtime, we propose to encode classified ontologies, represented by hierarchies of concepts, using numeric intervals as described in [CF03]. These hierarchies represent the subsumption relationships between all the concepts in the ontologies used in the repository. The main idea of the encoding is that any concept in a classified ontology is associated with a numeric interval. These intervals can be contained in other intervals but are never overlapping. The intervals are defined using a linear inverse exponential function:

$$linKinvexpP(x) = \frac{1}{p^{int(\frac{x}{k})}} + (x \bmod k) * \frac{1}{k} * \frac{1}{p^{int(\frac{x}{k})}}$$

where p and k are two parameters to be fixed. Regarding the scalability of this encoding solution, experiments show that, for p=2 and k=5 and a system encoding real numbers as 64 bit doubles, the maximum number of entries that we can have on the first level of the hierarchy is 1071 and the maximum number of levels that we can have on the first entries of a level is 462 levels.

Figure 4-6 taken from [CF03] shows an example of encoding a hierarchy of concepts with intervals.

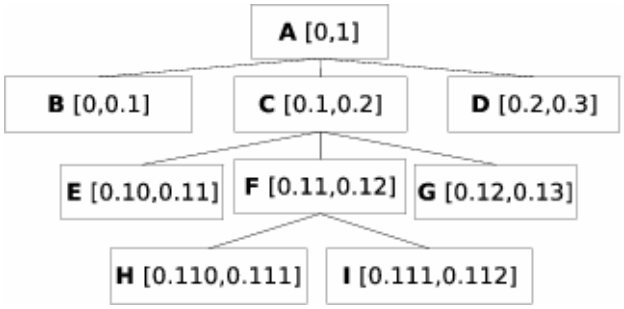

Figure 4-6: Example of encoding a class hierarchy

Under the assumption that the classified ontologies are encoded, and that service advertisements and service requests already contain the codes corresponding to the concepts that they involve, semantic service reasoning reduces to numeric comparison of codes. Indeed, to infer whether a concept C_1 represented by the interval I_1 subsumes another concept C_2 represented by the interval I_2, it is sufficient to compare whether I_1 is included in I_2. In order to ensure consistency of codes along with the dynamics and evolution of ontologies, service advertisements and service requests specify the version of the codes being used. We assume that services periodically check the version of codes that they are using and update their codes in the case of ontology evolution.

Semantic Service Advertisement and Matching

Based on the encoding technique defined in the previous section, we present an algorithm for matching a requested capability with a set of capabilities of networked services. Service capabilities could be added or deleted at any time from the existing set of capabilities. When a request comes, the algorithm tries to find a capability that best matches the request, minimizing the number of semantic matches performed with capabilities of networked services. At a pre-processing phase, the algorithm classifies capabilities of networked services and constructs directed acyclic graphs (DAGs) of related capabilities. These graphs are indexed according to the ontologies being used in the capabilities that they contain. The relationship between capabilities that we consider to construct a graph is given by the relation Match and

the function SemanticDistance. Specifically, if both Match(C_1, C_2) and Match(C_2, C_1) hold and SemanticDistance(C_1, C_2) = SemanticDistance(C_2, C_1) = 0, then C_1 and C_2 will be represented by a single vertex in the graph. For all the other cases where Match(C_1, C_2) holds, C_1 and C_2 will be represented in the graph by two distinct vertices with a directed edge from C_1 to C_2.

When a new service comes in the network, the set of capabilities that it provides are classified among the existing hierarchies. The algorithm of classifying new capabilities in the existing hierarchies is described in the following section.

When a request Req arrives, the algorithm first selects among the existing DAGs, graphs that contain services that are more likely to match the request. This is done using the indexes given to each graph, which correspond to the set of ontologies used by the capabilities of that graph. When a graph G is selected, the algorithm performs a matching between the request and the most generic capabilities of this graph. These capabilities are said to be more generic than other capabilities contained in their sub-hierarchy, because they provide a number of outputs that is greater or equal to the number of outputs of the other capabilities, and further because their provided outputs subsume the outputs of other capabilities (e.g., in Figure 4-5, the capability SendDigitalStream is more generic than the capability ProvideGame). These capabilities correspond to the capabilities represented by vertices of this graph that do not have predecessors, i.e., the set Roots(G). Similarly, we define Leaves(G) as the set of vertices in the graph G that do not have successors. If Match between Req and all the capabilities of Roots(G) does not hold, the group G is filtered out, and another group is selected. While, if the matching between the request and a capability C of Roots(G) holds, i.e., Match(C, Req) holds, we evaluate the semantic distance between C and Req. If the distance is equal to zero, C is selected, otherwise the algorithm tries to find a capability C from the successors of C such that SemanticDistance(C, Req) = min in (SemanticDistance(C_i, Req)), where C_i is a successor of C. The algorithm for answering a user request is presented in more details in the following sections.

Adding a new service advertisement

At a pre-processing phase, a set of DAG graphs are constructed and maintained. Each time a new service advertisement comes in the network, the graphs have to be updated with the set of capabilities provided by the new service. The algorithm of classifying the capabilities of a new service within a set of Graphs G_1, G_2,..., G_n is given below. For each capability C_i provided by the new service, the algorithm tries to find a graph G_i in which this capability will be integrated. A subset of graphs is pre-selected according to the ontologies being used by C_i. The algorithm first checks whether C_i can be inserted in the sub-hierarchy of one of the root nodes of G. This is done by verifying if there exists a node $Root_i$ in Roots(G_i) such that Match($Root_i$, Ci) holds. If Match($Root_i$, C_i) holds, then C_i will have a predecessor in G_i. The next step is to find this node, N_i, among the successors of the node $Root_i$, such that the Match(Succ(N_i), C_i) fails, and to draw an edge from C_i to N_i. Moreover, C_i could have a successor in G_i. Thus, the algorithm tries to find among the set Leaves(G_i) if there is a node $Leaf_i$ such that Match(C_i, $Leaf_i$). If Match(C_i, $Leaf_i$) holds, then C_i will have a successor in G_i. The next step is to find this node, N_i, among the predecessors of $Leaf_i$ such that Match(C_i, Pred(N_i)) fails, and to draw an edge from C_i to N_i. On the other hand, if Match(Rooti , C_i) does not hold, C_i will not have a predecessor in G_i. Nevertheless, C_i could have a successor in G_i. Thus, the algorithms checks whether there is a node $Leaf_i$ in Leaves(G_i) such that Match(C_i, $Leaf_i$) holds. The algorithm is given below:

input: C_1 , C_2 , ..., C_n the set of capabilities of the new service,

G_1 , G_2 , ..., G_m the set of existing graphs,

output: G_1 , G_2 , ..., G_k the set of graphs after the insertion of the new capabilities.

InsertCapabilities(capabilities)

```
For all the capabilities Ci in C1 , ..., Cn do{
   For all the graphs Gi in G1 , ..., Gm that use the same ontologies as Ci
   until the insertion of Ci do{
      For (Rooti in Roots(Gi )) do{
         If (¬Match(Rooti , Ci )) then{
            For (Leafi in Leaves(Gi )) do{
               If (¬Match(Ci , Leafi )) then
                  Fail;
               Else{
                  Test with Predecessors of Leafi
                  until ¬Match(Ci , Predj (Leafi ))
                  Draw an edge from Ci to Predj+1 (Leafi )
               }}
         }Else{
               Test with Successors of Rooti
               until ¬Match(Succj (Rooti ), Ci )
               Draw an edge from Succj−1 (Rooti ) to Ci
               For (Leafi in Leaves(Gi )) do{
                  If (¬Match(Ci , Leafi )) then
                     Fail;
                  Else{
                     Test with Predecessors of Leafi
                     until ¬Match(Ci , Predj (Leafi ))
                     Draw an edge from Ci to Predj+1 (Leafi )
}}}}}}
```

Figure 4-7 shows an example of inserting a capability, newC, in a DAG of capabilities, G. The first step (left part of the figure) is to match newC with capabilities from Roots(G) to find out whether newC will have a predecessor in G. Indeed, Match(C_1, newC) holds, which means that one of the successors of C_1 will be linked with newC, i.e., C_3. The next step (right part of the figure) is then to find out whether newC will have a successor in G. This is done by matching the capabilities in Leaves(G) with newC. Indeed, Match(newC, C_7) holds, which means that newC will be linked with one of the predecessors of C_7 , i.e., C_5 .

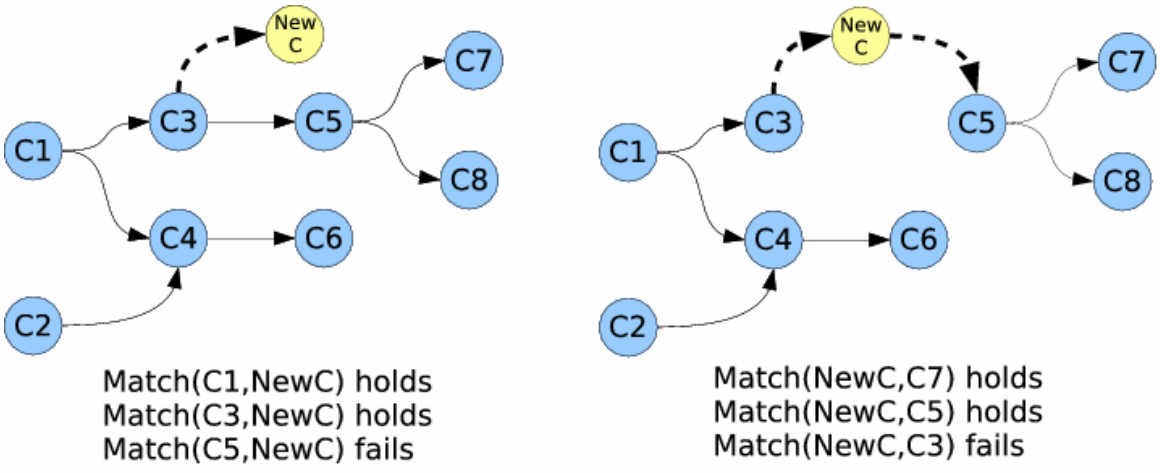

Figure 4-7: Example of inserting a capability in a DAG

Answering user requests

When a user request that contains a set of required capabilities comes, the algorithm below finds out a set of capabilities of networked services that best match the ones required by the user. More precisely, for each capability C_i in the user request, the algorithm tries to find a graph that may contain capabilities that match C_i. A graph G_i is selected if it is indexed with the ontologies used in the request and if there exist a node $Root_i$ in the set $Roots(G_i)$ such that $Match(Root_i , C_i)$ holds. In this case, a node that has the minimal semantic distance with C_i is selected from the successors of $Root_i$. The algorithm is given below:

inputs: a set of capabilities required in the service description $C_1 , C_2 , ..., C_n$,

a set of graphs $G_1 , G_2 , ..., G_m$,

outputs: a set of capabilities of networked services that match the capabilities given as input.

MatchService(requested service)

For all the capabilities C_i required in the service description do{

For all the graphs G_i in $G_1 , ..., G_m$ that use the same ontologies as C_i

until C_i is matched do{

For all $Root_j$ in $Roots(G_i)$ do {

If ($\neg Match(Root_i , C_i)$) then

Try with the next node in $Root(G_i)$

Else

Return $Succ(Root_i)$ from the successors of $Root_i$ such that

$SemanticDistance(Succ(Root_i), C_i)$ is minimal

}}}

An example of matching a requested capability with capabilities of networked services is given in Figure 4-8. In this figure, the requested capability NewC uses the ontology O_1 in its

specification. This allows to filter out DAG2, as it is indexed with only the ontology O_3. The next step is to match NewC with capabilities from Roots(DAG1) and Roots(DAG3), i.e., the capabilities C_1 and C_4. If the matching fails with one of these capabilities, we can infer that no capability will match newC in the corresponding graph.

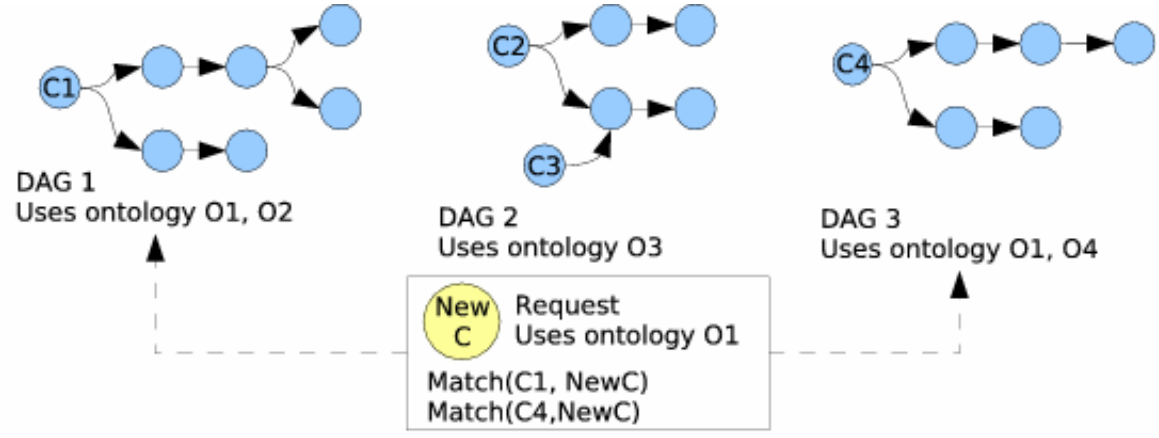

Figure 4-8: Example of matching a user's requested capability

The benefits of using this solution to match user's required capabilities with capabilities of networked services is to reduce the number of semantic matches performed to answer a query. Indeed, it is sufficient to perform a semantic match with a subset of the capabilities of networked services rather than performing a semantic match with all the capabilities hosted by the service repository. Furthermore, using the encoding of classified ontologies allows reducing the semantic reasoning to a numeric comparison of codes.

We have carried out extensive performance evaluation of our overall approach to efficient service matching and compared it with classic syntactic service matching, with highly encouraging results. Our results show that our solution provides better response time for the semantic matching of service capabilities than basic syntactic service matching. Furthermore, thanks to the indexing and structuring of the service repository, our solution is more scalable than a classic service repository. A detailed report on these results may be found in [BGI06b].

4.3.2 Service profile hierarchy-based matching

As pointed out in the OWL-S Technical Overview[12], an OWL-S service profile can be either a direct instance of the class Profile or an instance of any subclass of Profile. As a result, a service described in OWL-S (or in Amigo-S) can be classified into certain categories by letting its profile be an instance of any subclass of the class Profile. Based on this profile subclass hierarchy, profile matching can easily be done to determine how good the service category of the advertised service fits into the service category that the requested service demands. This Profile subclass matching complements the semantic matching based on service inputs and outputs, and can be combined with those to get more accurate results.

The Amigo functional capability hierarchy, as defined in the Amigo service modeling vocabulary ontologies (see Chapter 3), provides a classification hierarchy for intelligent home service profiles. This approach can be extended for classifying the capabilities introduced by a service profile and will be studied in a later version of this matching algorithm.

[12] http://www.daml.org/services/owl-s/1.1/overview/

Profile hierarchy-based matching is best used by combining it with input and output semantic matching for more accurate identification of a matching service. However, it is also useful in itself by providing means for an application to use the Amigo functional capability vocabulary to check the availability of services providing capabilities that belong to a specific category defined by Amigo vocabularies.

A profile match can be [P02]:

1. Exact match, i.e., the advertised profile class is the same as the requested profile class. According to the referenced article, if the requested profile class is an immediate subclass of the advertised profile class, this is considered as well an exact match.

2. Subsumption, i.e., the advertised profile class is any of the subclasses of the requested profile class.

3. Plug-in, i.e., the advertised profile class is a super-class of the requested profile class.

4. Fail.

The first two cases can be combined directly with other matching algorithms used by SD-SDCAE (see Section 4.3). The third case is more challenging, but may still be useful and even the most important case in the real world. For example, an application may request for a service for turning on a washing machine. There may be a direct match with the service provided by the washing machine itself; alternatively, there may exist a more advanced service efficiently controlling the energy of all home automation devices, provided by an energy conservation application.

As the processing of service capability hierarchies is a time and memory consuming process, several optimization algorithms for the ontology-based service matching have been recently proposed in the literature [CF03, SPS04]. We adopt the one introduced in [SPS04], as we believe it fits well in the service profile hierarchy matching. The rationale behind this approach is that the query for the service is the most time consuming process and may occur many times. Moreover, its response time is critical. Therefore, to speed up the process of query-response, it makes sense to process advertisements upon their publishing, i.e., to pre-compute the degree of match between advertisements and possible requests.

More specifically, we consider that the service repository is a dynamic list of services, which are published as records of <profile_name, concept, profile_link>, where: profile_name corresponds to the service name and is used for the further service binding procedure; profile_link specifies the location of the Amigo-S service profile description; and concept represents the category of the service according to the service capability hierarchies maintained by the matchmaker. Upon arrival of an advertisement, the pair of <profile_name, profile_link> is saved into the repository. At publishing time, the matchmaker also annotates each concept in the service capability hierarchies with information which specifies to what degree (i.e., exact, subsume, etc.) any request pointing to this concept would match the profile category found in the advertisement. At query phase, the concept which represents the queried profile category is compared with the information pre-computed at publishing time.

For example consider the services related to an AV scenario which typically involves the multimedia content transfer. The part of the related service capability hierarchies is presented in Figure 4-9.

Let us assume that several services have been published in the repository providing media server and media renderer capabilities according to specified records, as follows (URL data is omitted as it is not essential for this example):

<HelixServer, MediaServer, url>

<WindowsMediaServer, MediaServer, url>

<AllegroMediaServer, MediaServer, url>

<Philips37PF7320A_LCD_HDtvRenderer, MediaRenderer, url>

<Dell109721_laptopRenderer, MediaRenderer, url>

<AmigoAVControl, HomeAVControl, url>

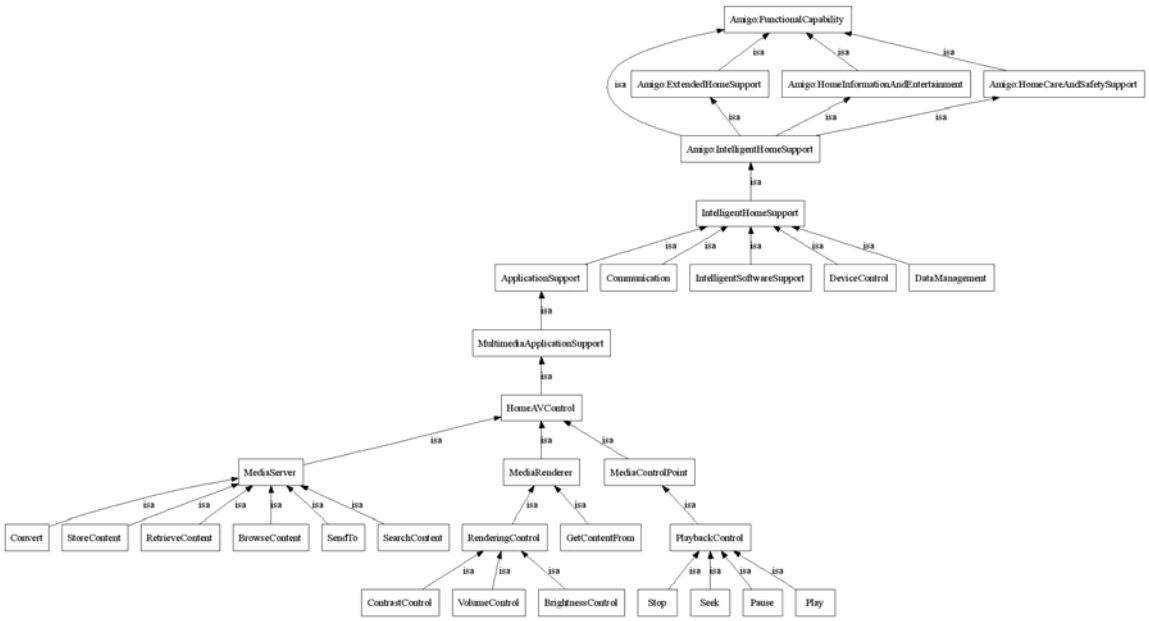

Figure 4-9: The part of the service capability hierarchy related to the Multimedia Application support

Then the concepts maintained by the matchmaker will be updated with the following information specifying the respective degrees of match:

HomeAVControl (<AmigoAVControl, exact>, <HelixServer, subsum>, <WindowsMediaServer, subsum>, <AllegroMediaServer, subsum>, <Philips37PF7320A_LCD_HDtv, subsume>, <Dell109721_laptop, subsume>)

MediaServer (<AmigoAVControl, exact>, <HelixServer, exact>, <WindowsMediaServer, exact>, <AllegroMediaServer, exact>)

MediaRenderer (<AmigoAVControl, exact>, <Philips37PF7320A_LCD_HDtv, exact>, <Dell109721_laptop, exact>)

MediaControlPoint (<AmigoAVControl, exact>)

Let us now assume that the requester/application is looking for the services necessary to present multimedia content for the user. This task would require some Media Servers to retrieve the content and some Media Renders to present this content to the user. Then, it is natural to specify the profile category for the query represented by the concept HomeAVContol. If we agree to use only ‘exact’ and ‘subsumption’ degrees of match, then, by accessing the above pre-computed service match degrees, the links to all available fitting service profiles will be returned by the matchmaker. Then, further processing may be carried out with more precise matching algorithms, for example, capability matching; or, filtering may be performed with the context information specified by the client and provided by the service.

Accordingly, if the application is looking only for media renderers, the concept specified will be the ServiceRenderer, and three services, i.e., Philips37PF7320A_LCD_HDtvRenderer,

Dell109721_laptopRenderer and AmigoAVControl (as multi-functional service), will be selected.

The matching algorithm can be implemented as simple keyword search within the functional capability hierarchy, when the vocabularies used are relatively small. There are also several ways to use this simple profile matching approach in situations where more advanced SD-SDCAE matching services are not available.

The service capability hierarchy-based matching proof-on concept prototype has been implemented over Jena OWL-S framework query mechanisms. Its testing with the developed simulator (see Section 3.3) is currently in progress.

Using the OSGi platform, the semantic service capability hierarchy-based service discovery can be implemented through the standard OSGi bundle mechanism. The semantic service discovery bundle may work in cooperation with the existing OSGi framework LDAP-based service registry mechanism, returning a discovered service implementation class name for direct binding.

4.3.3 Context-aware service selection

The Context-Aware Service Selection (CASS) functionality is part of the overall SD-SDCAE offering of Amigo. In Figure 4-1, this functionality can be seen as Step 5. In the following sections, we provide a detailed design of the Context-Aware Service Discovery (CASD) functionality which is more than just CASS. Note that in this design, we do not yet consider the integration of the CASD's functionality within SD-SDCAE but, rather, start with a more fundamental usage scenario, leaving the integration aspects as further work.

Usage scenario

A client needs to obtain a reference to the most appropriate service available within its vicinity. The client has a semantic description of the service that it wishes to locate according to a specified "matching constraint", and also a specification of the conditions that the client wishes to match upon. In this scenario the client wishes to locate the "cheapest" appropriate service.

Thus, the client issues a request to the CASD service, requesting it to obtain the cheapest matching service. The CASD processes the request (using an algorithm outlined below) and returns the most appropriate match, taking both the semantic service description and the matching constraint into account. If the client wishes to be notified when a different service becomes "more appropriate", then instead of issuing a "single" lookup request to the CASD, it issues a persistent lookup request. This results in the client being told the current most appropriate matching service and also being notified when a different service becomes the most appropriate. The client can cancel the callbacks if it wishes to.

Externally visible interface

The CASD publishes an externally visible service interface which contains the following methods:

```
public AmigoRef lookup(SemanticServiceDescription requiredServiceDescription, AmigoRef[] clientContextSources, String casdSelectionIdentifier);

public AmigoRef subscribePersistentLookup(SemanticServiceDescription requiredServiceDescription, AmigoRef[] contextSources, String casdSelectionIdentifier, AmigoRef callBackServiceReference);

public void unsubscribePersistentLookup(AmigoRef callBackServiceReference);
```

The **lookup** method is invoked by the client to perform a one-off context-aware service discovery. The client provides the following parameters:

- **SemanticServiceDescription** – a description of the service the client is looking for (using the Amigo-S service description language, see [Amigo-D3.2].
- **AmigoRef[]** – an array of AmigoRefs that point toward the context sources that provide contextual information about the client.
- **casdSelectionIdentifier** – a statically defined string that is used by the client to indicate what type of "contextual optimization" should be performed by the CASD.

We define **casdSelectionIdentifier** to take one of the following values:

```
public static final String CURRENT_PREFS = "…";
public static final String CLOSEST = "…";
public static final String CHEAPEST = "…";
public static final String FASTEST = "…";
```

The **subscribePersistentLookup** method is called by a client when it wishes to obtain a reference to the most appropriate service and also would like to be informed when the service is no longer the most appropriate according to the specified criteria. In this case, the client is called-back with the new best matching service. The client is required to supply the same parameters as for the lookup call, but also one additional parameter:

- **callbackServiceReference** – an AmigoRef that points toward the AmigoService that should be called when the most appropriate matching service changes.

The method returns an AmigoRef which is used as a parameter to the last publicly available method which is **unsubscribePersistentLookup**. This method allows a client to indicate that it is no longer interested in receiving callbacks from the CASD when the most appropriate matching service changes.

CASD Algorithm

The CASD performs the following sequence of steps when servicing a lookup request:

1) Interact with semantic service discovery (Step 4 in Figure 4-1) to obtain a set of semantically matching services.
2) For each of these services, obtain a reference to their context source(s). Interact with context sources(s) to obtain the current context of the service.
3) Interact with the client's context source(s) to obtain the client's current context.

Note: an optimization could be that the CASD only obtains context (from the services and client) which is appropriate for the type of service matching criteria specified in the client's request, e.g., when obtaining the "closest" service, only location context is relevant.

4) Depending on the type of service matching criteria specified by the client, hand off the context information of the client and the service(s) to a ContextComparator service. This service will process the service and client contexts according to the service selection directive supplied by the client and return the same set of input services as an ordered list of suitability.

Context Comparator Service

This module (CCS) is used by the CASD to compare and evaluate context information. It takes context information and a selection directive and returns an ordered set of services. It achieves that as follows:

In the case of matching on "Fastest" and "Cheapest", the CCS pushes the contexts of the matching services into Jena (as RDF documents) and queries it (using a SPARQL query) to find the (id of the source of) matching contexts, ordered for example from lowest to highest. We then determine the AmigoRef of the services which the contexts refer to and return this ordered list of AmigoRefs back to the calling client. The client can then convert the "first" AmigoRef into an AmigoService and invoke the service it was looking for.

Note that this approach does not work entirely for the "closest" situation, since we need to be able to relate absolute locations towards a "map" of the home, etc. This will require a more complex context comparator that can actually convert GPS locations into more meaningful data. This is ingoing work and will require us to use the Task 3.9 Location Service and also the (still missing) Amigo Configuration service.

4.3.4 QoS-aware service selection

QoS-aware service selection is depicted as Step 6 in Figure 4-1 of the overall SD-SDCAE functionality. In the following, we address the case where the resources of a server should be shared among concurrently submitted service requests, and, thus, a resource-aware and QoS-aware allocation should be performed. Note that in this design, as with context-aware service selection, we do not yet consider the integration of this functionality within SD-SDCAE, but, rather, start with a more fundamental usage scenario, leaving the integration aspects as further work.

A plethora of services will eventually be deployed in the Amigo home. Many of these services will be offering similar functionality. For serving a specific service request of a user, the Amigo home middleware should be able to select the most suitable one among services with similar functionality and similar IOPE (Inputs-Outputs-Preconditions-Effects) parameters, all addressing the user requirements. Thus, the selection process will depend heavily on these functional parameters (Step 4), and will also consider the number, features and identity of the services that are already selected and/or available. Additionally, another issue that may have a significant impact on the performance of the Amigo system is the fact that multiple users may concurrently submit several service requests to a *server* residing on a networked home device, expecting the services to be delivered at the same time, each being compliant with the user preferences. However, as the resources of the Amigo middleware and the server in question will not be unlimited (e.g., with regard to capacity, bandwidth, processing capabilities, storage), a service selection tool must be established to decide on the services that will eventually be delivered, and on their configuration and properties, so that the system resources are used in the best possible manner, while users enjoy the services that address their requirements as much as possible.

Hereafter, an illustrative example concerning outdoor services (i.e., services that use the incoming network bandwidth and are delivered to a user device via a Home Gateway) is described, which aims to clarify the grounds of this service selection optimization problem. Let’s assume that in the Amigo networked home there is a LAN established and that there is a 1024 kbps DSL line shared by all networked devices. Let’s further assume that there is an incoming request of an Amigo user for playing a game on the Internet. This gaming service is offered in various versions that require different bandwidth rates (e.g., 56 kbps, 128 kbps, 256 kbps and 384 kbps). In order for the Amigo middleware to decide which service version is the most appropriate, it has to consider several parameters. First, it has to filter out the service versions that do not address the user requirements (e.g., with regard to bandwidth, price, image resolution). Then, it has to discover if other service requests are in place and which

ones, and consider the resources that will potentially be consumed by the relevant service deliveries. At this point, the system needs to select the services to be delivered considering the service features, user requirements and resource constraints. Even if the system resources are enough to satisfy all service requests, the service selection process is still necessary in order to ensure that the Amigo users' objectives and needs are efficiently fulfilled.

The proposed QoS-Aware *Service Selection Tool* (QASST) will be provided by the Amigo Middleware, which will reside on the networked home devices. Among these devices, there is the Home Gateway that enables the Amigo Home to be connected to the Internet. QASST provides service selection mechanisms adequate not only for requests concerning outdoor services, but also for services concerning in-home activities, e.g., content delivery services that reside on Amigo home devices. Certainly, it is assumed that a base service discovery mechanism is available (Step 4), enabling the discovery of both kinds of services (indoor and outdoor) that match the functional requirements of the service request.

In the context of QoS-aware service selection, a client submits its service request to the QASST. The QASST uses the base service discovery mechanism to identify the services matching the functional requirements of the service request received. Then, it retrieves (from the User Modeling and Profiling Service – UMPS of the WP4 Intelligent User Services layer) the user's QoS preferences. Subsequently, the matching mechanism of QASST takes control and filters out the services that do not address the user's QoS requirements. The actual service selection process depends on whether this service request involves only indoor or also outdoor services.

In case only indoor services match the user requirements, a selection process takes place based on a linear utility function. This utility function is expressed by the weighted sum of the normalized values of the QoS parameters that a user prefers to have as high as possible (for example availability, capacity, etc.) minus the weighted sum of the normalized values of the QoS parameters that a user prefers to have as low as possible (for example response time, error rate, cost, etc.). The matching service maximizing the value of this utility function will be finally selected as the most suitable one. This utility function can be expressed as follows:

$$U = \sum_{i=1}^{l} w_i x_i - \sum_{j=1}^{m} w_j y_j \qquad (1)$$

where l (m) is the number of the QoS parameters that are preferred to be as high (low) as possible, x_i (y_j) are the normalized values of these QoS parameters, and w_i (w_j) are the normalized weights of these QoS parameters based on the user's QoS preferences for each service type. Thus, the following equation must hold: $\sum_{i=1}^{l} w_i + \sum_{j=1}^{m} w_j = 1$. It should be mentioned that the normalization of the QoS parameters is derived by dividing the actual values of the QoS parameters offered by the services with the maximum acceptable or possible value these parameters are limited by.

The necessary interactions of the QoS-aware Service Selection Tool components with external actors as well as other Amigo components are depicted in the sequence diagram of Figure 4-10 for the indoor service selection process.

On the other hand, if outdoor services match the requirements of the service request, the selection process is more complex. In Figure 4-11, the suggested service selection process for an outdoor service is illustrated, along with the involved modules/actors. In step (1), an additional service request (Req. 5) is submitted by an Amigo User. Note that at that time, there are already four on-going sessions for services that were previously requested (Req. 1 – Req.

4). In step (2), the client selects to use the QoS-aware Service Selection mechanism. In step (3), the QASST initially identifies a set of references for the available services that address the functional requirements of the new service request (which are discovered through the base service discovery mechanism). Then it determines the services that will be eventually selected for serving all service requests of the Amigo Users (i.e., Req. 1 – Req. 5), re-evaluating the service selection performed upon the reception of previous requests.

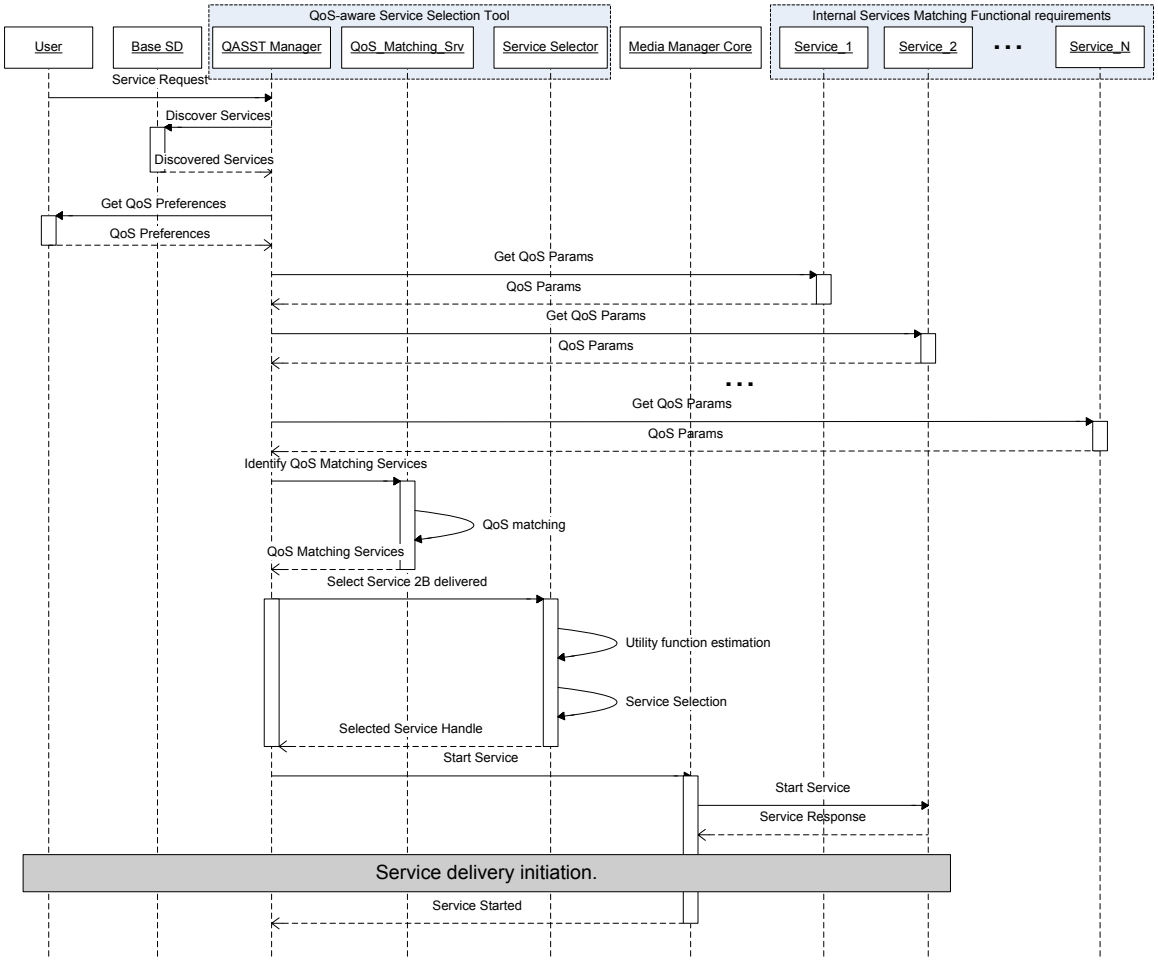

Figure 4-10: The Service Selection sequence diagram for indoor services

Based on the aforementioned analysis for the outdoor services' requests, it is clear that not all the requests can be served in a real home environment, where the available resources (e.g., bandwidth, capacity, etc.) are limited. Thus, we will quite often face the problem of not having enough resources to address all the users' requirements. Thus, not every service request is always feasible to get served, or at least not in the most preferable service version for the users.

The proposed QASST will depend heavily on priorities. We adopt a priority-based selection model, as it is desired: (i) to serve as many requests as possible, in order to satisfy the majority of Amigo users; and (ii) to firstly serve all requests carrying a higher priority. For example, safety-related requests should be served in any case, and should thus have the highest possible priority. The proposed priority model consists of two levels. The most important level (first level) depends on the kind of service that is requested (e.g., safety, gaming, information, entertainment, etc.). Of course, safety-related services are assigned with

higher priority than entertainment-related services. So, in Figure 4-11, request 5 will be served first with regard to requests 1 or 2. The second level (less important level) of the proposed priority model depends on the person who submits the request. In this case, parent-originated service requests, for example, are assigned with higher priority than children's requests.

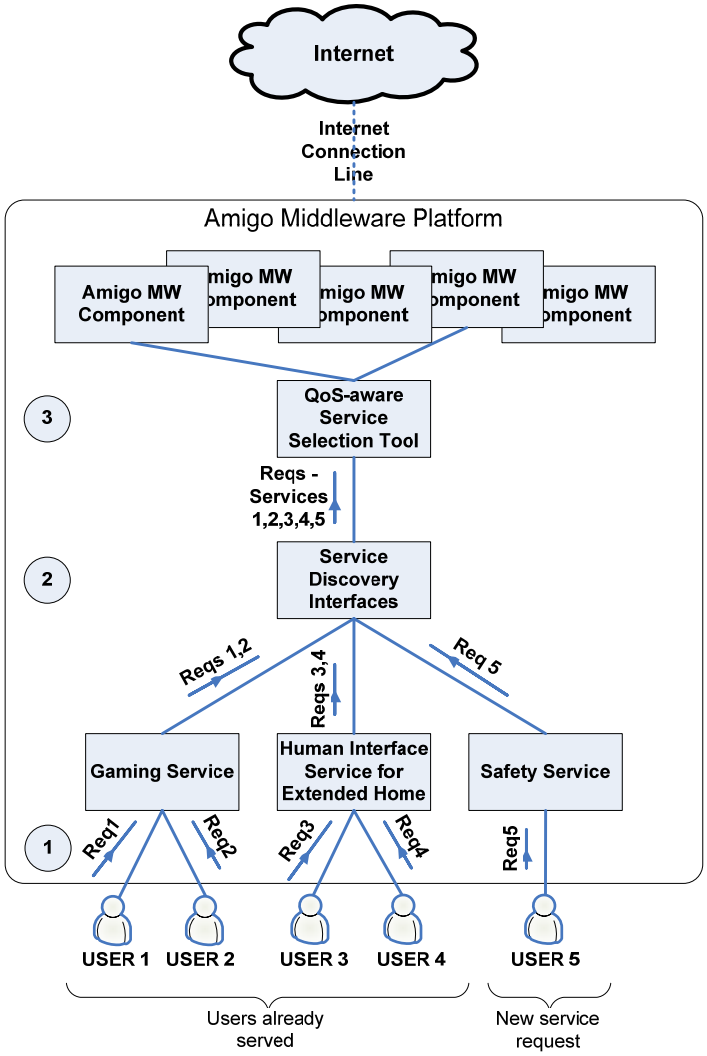

Figure 4-11: The selection process for outdoor services

The QASST follows a straight-forward approach in order to determine the priorities of the service requests. To accomplish this, a binary representation of the number l_i of the priority instances of each level *i* is used. The number of bits assigned at each level is estimated based on the population of the priority instances of this level. For example, if we have 19 instances under priority level 2, then the binary representation of this level's priority will require 5 bits (19:10011). Thus, the combined priority (*combined_prio*) of each service request is calculated based on the following equation:

$$combined\_prio = \sum_{i=1}^{k} prio_i \, 2^{A_i} \;,\; A_i = \sum_{j=i+1}^{k} a_j \qquad (2)$$

where $prio_i$ is the service priority for level i, a_i represents the number of bits required for the individual priority binary representation of level i, A_i indicates the number of bits required for the combined priority binary representation of level i, and k represents the number of priority levels.

The above are illustrated in the following example: Consider the case of four main priority levels. The first level has 23 instances ($\rightarrow a_1 = 5$), the second has 35 ($\rightarrow a_2 = 6$), the third has 12 ($\rightarrow a_3 = 4$), and the last has 19 instances ($\rightarrow a_4 = 5$). Based on Equation (1), we have: $A_1 = 15$, $A_2 = 9$, $A_3 = 5$ and $A_4 = 0$. This model is extendible, so that additional priority levels can be distinguished if necessary.

The steps that will be followed in the services' selection process concerning the outdoor services are:

1. Initially, a new request is submitted, and QASST identifies a set of references for the services that address the user's requirements concerning the functional requirements of the specific service request through the base service discovery mechanism.
2. The QASST then performs further filtering aiming to exclude the services that do not match the QoS requirements of the service request (e.g., maximum service cost requirement).
3. The remaining services address all the user requirements, and they are recorded into a Service Registry (SR) as matching services.
4. A check is made in order to identify whether the services that have been selected and have started after the last enforcement of the service selection algorithm (right after the reception of the previous service request) are still running. The ones that have stopped are removed from the SR, while the others remain recorded as running services.
5. A check is made in order to identify whether new priorities have been added/modified or old priorities have been removed.
6. The priority of the new service request is calculated, while – if necessary – the priority of the requests for all on-going service sessions are re-estimated.
7. For the new request and each of the old requests related to on-going sessions, the minimum bandwidth service is identified, which addresses the requirements of the user request. Thus, one service per such request is identified. These services are then ordered based on their priority. The selection of requests that will eventually be served is made based on the overall bandwidth that can be supported by the gateway, i.e., higher priority requests are served (Part I).
8. In case a service that is already running has to be terminated because a new service request having higher priority has been received, a "Terminate Service Delivery" request is made.
9. For the selected requests to be served, the Service Selection Algorithm is triggered again to identify the services that will serve these requests. The service selection process now aims to minimize the overall cost for the users (Part II).
10. Once the services to be provided are selected, first, the ones that are currently running but have not been selected are terminated, and, second, the ones that are not currently running but have been selected are started.

The aforementioned steps of the selection process of outdoor services are illustrated in the state diagram of Figure 4-12.

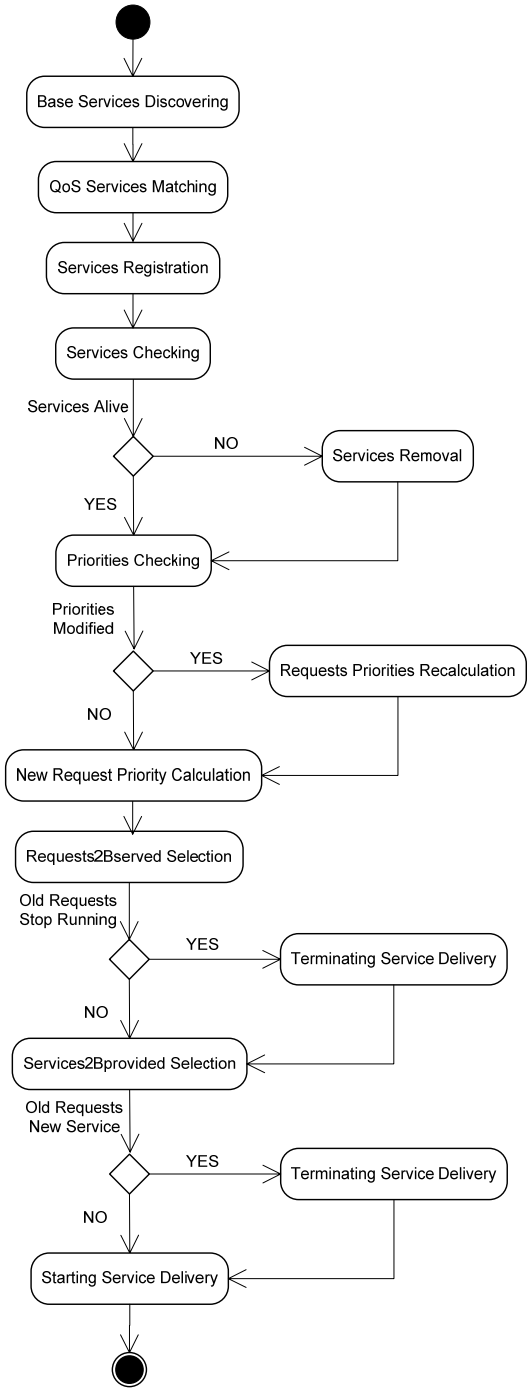

Figure 4-12: The Service Selection process state diagram for outdoor services

The necessary interactions of the QoS-aware Service Selection Tool components with external actors as well as other Amigo components are depicted in the sequence diagram of Figure 4-13 for the outdoor service selection process.

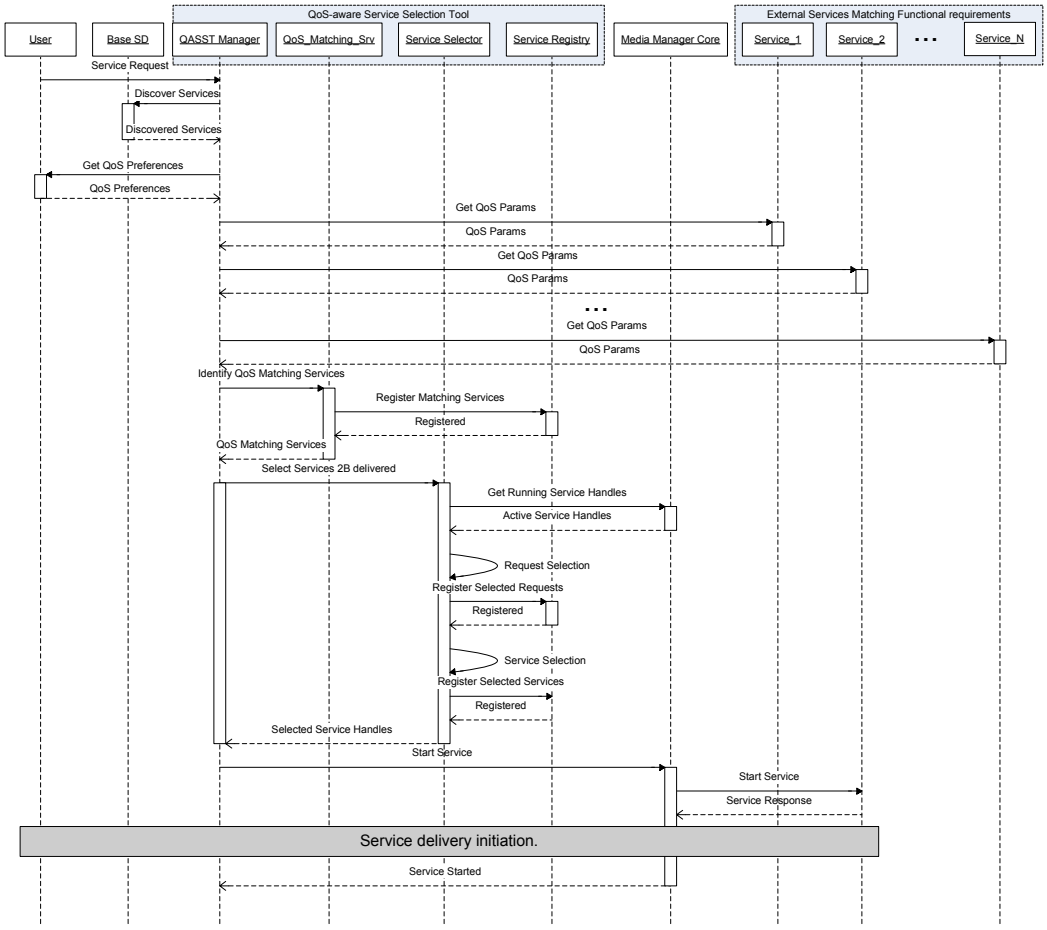

Figure 4-13: The Service Selection sequence diagram for outdoor services

The presented process aims to solve the service selection problem that can formally be stated as follows: Given *N'* service requests from Amigo Users and given all the available services that address their requirements, select the most appropriate set of services to be delivered so that the maximum possible number *N* of requests is served, the total priority-weighted cost is minimized, while the overall bandwidth of the selected services does not exceed the one provided by the established infrastructure. This can be reduced to the following linear programming problem:

Objective function: min $$\sum_{i=1}^{N}\left[\sum_{j=1}^{M_i}\left[(c_{ij} / combined\_prio_{ij})S_{ij}\right]\right] \quad (3)$$

Restrictions: $$\sum_{i=1}^{N}\left[\sum_{j=1}^{M_i} b_{ij} S_{ij}\right] \le B \quad (4)$$

$$\sum_{j=1}^{M_i} S_{ij} = 1,\ i = 1,2,...,N \quad (5)$$

$$S_{ij} \in \{0,1\},\ i = 1,2,...,N\ ,\ j = 1,2,...,M_i \quad (6)$$

where *N* is the overall number of concurrent service requests, M_i is the overall number of the services that address the requirements of service request *i*, S_{ij} is the decision variable for service *j* that addresses the requirements of service request *i,* b_{ij} is the required bandwidth by service *j* for service request *i*, c_{ij} is the corresponding service cost, and $combined\_prio_{ij}$ is the combined priority of service *j* for service request *i.*

This is a minimization problem seeking to minimize the overall cost. The restrictions of this problem suggest that: the overall bandwidth of the selected services does not exceed the available bandwidth B (Equation 4), every request is served (Equation 5), and the decision variables are Boolean (Equation 6), i.e., $S_{ij} = 1$ in case service *j* is selected to serve service request *i* or $S_{ij} = 0$ otherwise.

It stands that $N \le N'$. The number N of requests that can be served simultaneously can be estimated based on the priority model defined above. The estimation process is as follows: First, the priority of each service request is calculated. Then, the service requests are ordered based on their priority (i.e., *i=1* for the highest priority and *i =N'* for the lowest). For each service request *i*, the lowest bandwidth service *l* is selected in the set of services that address the request's requirements. N is provided by the following equation:

$$N = \left\{\max k : \left[\sum_{i=1}^{k} b_{il}\right] \le B\right\} \quad (7)$$

Of course, in case $\sum_{i=1}^{N'} b_{il} \le B$, then *N = N´*.

After having identified an initial solution to our service selection problem, we will refine our solution in order to reduce the overall priority-weighted service delivery cost. The problem is currently being studied.

4.4 Service composition

After service registration & advertisement in a repository and discovery & selection of services that match the requirements of a user application – Steps 1 to 6 of Figure 4-1 discussed in the previous sections, service composition is carried out within the SD-SDCAE approach (Step 7). Service composition enables increased availability of functionalities and new complex functionalities in the Amigo home. Indeed, a functionality can be provided by a single service or by a combination of several ones that are potentially not aware of each other. We introduce

service composition from an orchestration viewpoint, where the application (the task on the user device, see Figure 4-2) combines functionalities offered by several remote services.

In the following three sections, we propose three – partly complementary, partly alternative – approaches to service composition. In Section 4.4.1, we introduce an approach to integration of services modeled as conversations towards realizing a user task also modeled as an orchestration. Then, Section 4.4.2 presents the event strategy rule reasoning approach towards specifying the desired service composition for a situation and its variations in the Amigo home. Finally, section 4.4.3 introduces a scripting language for event-driven service composition (see Figure 4-2). Combining these approaches to service composition within SD-SDCAE is still an open issue and is considered as future work.

4.4.1 Conversation matching and integration

For the dynamic composition of services in AmI environments, and more particularly in the Amigo home, following the semantic service discovery and selection of networked service candidates, we introduce an algorithm (COCOA-CI) that performs the dynamic composition of the selected services towards the realization of a target user task. Early work on this algorithm was presented in [Amigo-D2.1]. The distinctive feature of COCOA-CI is the integration of services modeled as conversations to realize a user task also modeled as an orchestration. This provides a means to deal with the diversity of services in the Amigo home. Indeed, integrating service conversations for the realization of a user task enables the same user task to be performed in different environments under several composition schemes (e.g., using a different number of services with different conversations). Thus, the realization of the task's orchestration is adaptive according to the specifics of the environment in terms of available networked services and their provided conversations. Moreover, COCOA-CI enforces a valid consumption of the composed services as their conversations are fulfilled.

COCOA-CI operates on services' conversations and the task's orchestration described in Amigo-S, which actually adopts the OWL-S process model. COCOA-CI assumes a specific model for an Amigo-S conversation/orchestration, which is presented in the following section.

Amigo-S service conversation and task orchestration model

In this model, the central concept is the notion of *capability*. A capability is a functionality provided by a service or requested by a task and is described as a set of IOPEs. A capability is realized by either an OWL-S atomic or composite process[13]. We model both networked services and user tasks as a set of capabilities coordinated in the form of a workflow. This workflow is described using the OWL-S process model. Within the task description, the capabilities are *abstract*, i.e., they do not refer to any specific networked service, as these capabilities have to be dynamically provided by the AmI environment and may be realized by either atomic or composite processes of networked services. Indeed, this depends on the service implementation. The same capability can be developed as a single client/service interaction or as a sequence of client/service interactions (e.g., the Amazon Web service[14] has been described in the form of both a single atomic process and a complex conversation). Finally, some capabilities of the user task conversation can be specified as *correlated*. A set of correlated capabilities of the task must be performed by a single networked service because of the presence of internal/hidden dependencies between these capabilities (e.g., the reservation and the payment of a hotel room, for which it would be inconvenient to book a room in a hotel and to pay another hotel for this room). All the other parts of the user task are considered as independent from each other, yet with external/explicit dataflow. For the realization of these

[13] By default, if no capability is specified in the service description, atomic processes are considered as the provided capabilities.

[14] http://www.daml.org/services/owl-s/examples

independent parts of the user task, COCOA-CI allows the interleaving of multiple services' conversations.

Based on the above service conversation and task orchestration model, we introduce COCOA-CI in the following section.

COCOA-CI

To integrate the conversations of the selected services towards realizing the user task's orchestration, COCOA-CI translates both conversations and orchestration into finite state automata. Thus, the conversation integration problem becomes a finite state automata analysis problem. Further details about this modeling based on automata may be found in [Amigo-D2.1, BGI05].

COCOA-CI first integrates all the automata of selected services in one global automaton. The global automaton contains a new start state and empty transitions that connect this state with the start states of all service automata. The automaton also contains other empty transitions that connect the final states of each service automaton with the new start state. Consider the automaton representing the conversation of the target user task depicted in Figure 4-14, left higher corner, and the automata representing the conversations of the selected services at the right lower corner. In this figure, all the automata of the selected services are connected in a global automaton, in which all the added transitions are represented with dashed lines.

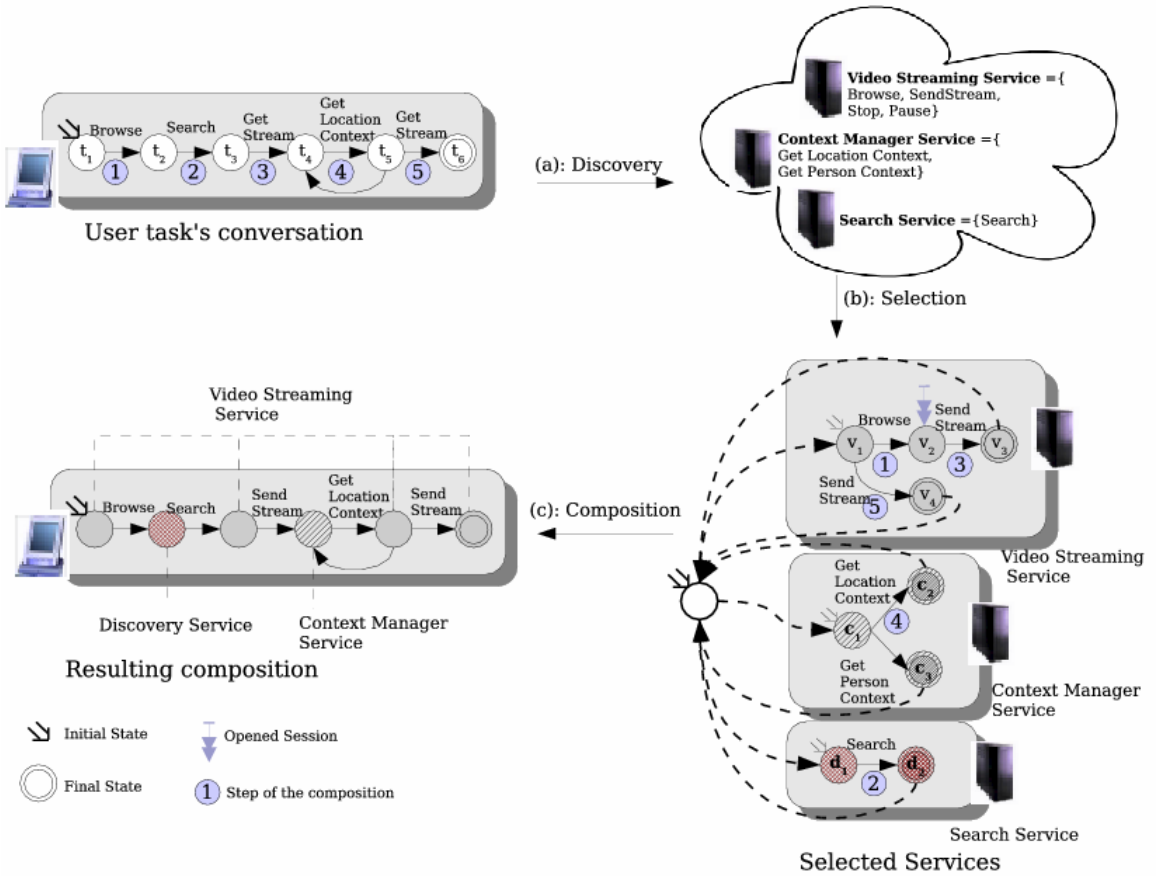

Figure 4-14: Conversation integration

The next step of COCOA-CI is to parse each state of the task's automaton, starting with its start state and following its transitions. Simultaneously, a parsing of the global automaton is carried out in order to find for each state of the task's automaton a state of the global automaton that can *simulate* it; specifically, a task's automaton state is simulated by a global automaton state when for each incoming symbol[15] of the former there is at least one semantically equivalent[16] incoming symbol of the latter. For example, in Figure 4-14, the state t_1 of the task's automaton can be simulated by the state v_1 of the global automaton because the set of incoming symbols of t_1 is a subset of the set of incoming symbols of v_1.

COCOA-CI allows finding service compositions with possible interleaving of conversations of the involved services. Indeed, this is done by managing service sessions. A service session characterizes the execution state of a service conversation. A session is opened when a service conversation starts, and ends when this conversation finishes. Several sessions with several networked services can be opened at the same time. This allows interleaving the interactions with distinct networked services. Indeed, a session opened with a service A can remain opened (temporary inactive) during the interaction of the client with another service B. An example of managing sessions is given in Step (1) of the composition of Figure 4-14. In this step, the capability *Browse* of the task's automaton has been matched against the capability *Browse* of the global automaton. The next step is to find the capability *Search* of the task's automaton (Step (2)). However, this capability is not available in the *Video Streaming Service*. This leads to open another session with the *Search Service*, as this service provides the sought capability. In Step (3), after matching the capability *Search*, the capability *Get Stream* is sought. A semantically equivalent capability, i.e., the *Send Stream* capability, is accessible in the *Video Streaming Service* from the previously opened session.

An important condition that has to be observed when managing sessions is that each opened session must be closed, i.e., it must arrive to a final state of the service automaton. During the composition process, various paths in the global automaton, which represent intermediate compositions, are investigated. Some of these paths will be rejected during the composition, while some other will be kept (e.g., if a path involves a service in which a session has been opened but never closed, this path will be rejected).

COCOA-CI gives a set of sub-automata of the global automaton that conform to the task's automaton structure (one such sub-automaton is depicted in the left lower corner of Figure 4-14). Each of these automata is a composition of networked services that conforms to the conversation of the target user task, further enforcing valid service consumption. The last step is to select arbitrarily one among the resulting service compositions, as they all conform to the target user task. We are further working on the definition of a benefit function adapted to the requirements of the Amigo environment (e.g., by taking into account an estimation of the availability of the composed services [LI05]) which will allow the selection of the most effective composition among the eligible ones. Using the service composition that has been selected, the Amigo-S process model description of the user task is complemented (concretized) with information coming from the composed services. Specifically, each capability of the user task is replaced with the corresponding capability of a networked service. This capability may correspond to either an atomic or a composite process of the networked service. In the former case, the capability is performed in a single client/service interaction, while in the later it will be performed in a sequence of client/service interactions.

We have carried out extensive performance evaluation of the COCOA-CI approach to service composition based on conversation integration, with highly encouraging results. We have compared the response time of COCOA-CI against the time spent for the XML parsing of

[15] Incoming symbols of a state correspond to the labels of the next transitions of this state.

[16] We recall that the equivalence relationship between capabilities is a semantic equivalence that has already been checked by the semantic service discovery algorithm.

services and task descriptions, which is inherent to the use of Web services and semantic Web technologies. Results show that in realistic cases COCOA-CI overhead is negligible compared to XML parsing. A detailed report on these results may be found in [BGI06a].

4.4.2 Rule & strategy-based reasoning and integration

An example of a challenging situation for SD-SDCAE is when the use of a high-level service that provides plug-in match for a discovery request is more desirable than an exact match of the discovery request using profile matching (see Section 4.3.2). Adaptive service composition in SD-SDCAE supports the application development by providing to the application developer the means to define one or more intelligent strategies to solve this type of situations.

The event strategy rule reasoning (ESRR) support of SD-SDCAE helps to specify the desired service composition for a situation in the Amigo home. The current situation can be checked with rules using a RDQL query on the available context information, and the resulting composite service can be specified semantically with the Amigo-S language. An optimized solution for resource-constrained devices uses the service discovery interface of SD-CAE to find services (see Section 4.3) and the WP4 Context Management Service (CMS) component for obtaining associated context information. In this first iteration, the event capabilities are left out of the ESRR realization.

An example of how ESRR and profile matching can be used in adaptive service composition is given in the following:

1. The adaptive application requests a high-level Amigo functional capability using the profile-based matching support of service discovery.
2. Discovery returns a set of service descriptions, using the Amigo functional capability hierarchy for profile matching.
3. The application can use the ESRR support of adaptive service composition to select the most suitable services and the associated composition (i.e., realization of the application that composes its functionality from available services).

In its implementation, the approach combines several software design patterns into one, most importantly:

- ECA (Event Condition Action) rules for reasoning about the situation in Amigo home.
- Strategy pattern for representing multiple composition choices.
- Composite pattern for hierarchic reasoning support.

The ECA paradigm is applied to the composition **rules,** which consist of a condition and an action part. The event of the ECA paradigm is shared by a set of rules that are composed by an ESRR. Using the composition rules, an application developer can define an assumption of current situation in the Amigo home in the condition part, and express an associated service composition in the action part (see Figure 4-15). In the first implementation of rules, the condition part is restricted to only specifying a set of desired functional capabilities that the associated composition in the action part needs to have available in the Amigo home.

The **condition** part of a rule is a query related to context in the Amigo home. The RDQL query language can be used to simulate these queries. In this first iteration, the type of queries is restricted to inputs and results of SD-SDCAE service discovery requests.

Actions define a composition that suits the result of successful condition query. In this first iteration, the action part is either a Java class that implements the composite service or an Amigo-S composite service description, possibly parameterized with the results of the condition query.

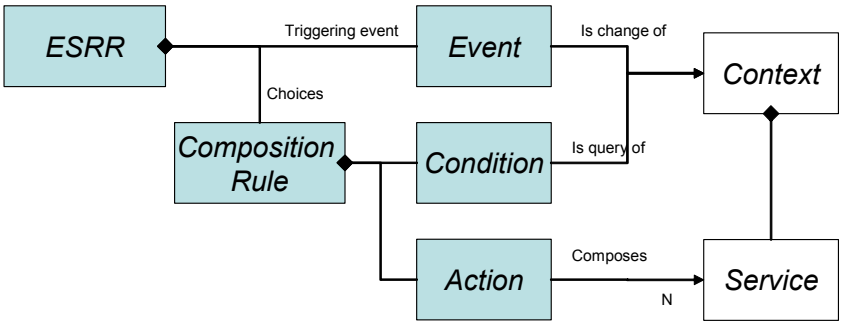

Figure 4-15: ECA paradigm applied to ESRR

Composition strategies are mutually exclusive so that one from a set of strategies can be selected at a time (See Figure 4-16). A strategy is a number of composition rules, so its instances are here referred to as strategy rules. The list of strategy rules in ESRR is ordered and all the rules share the same event.

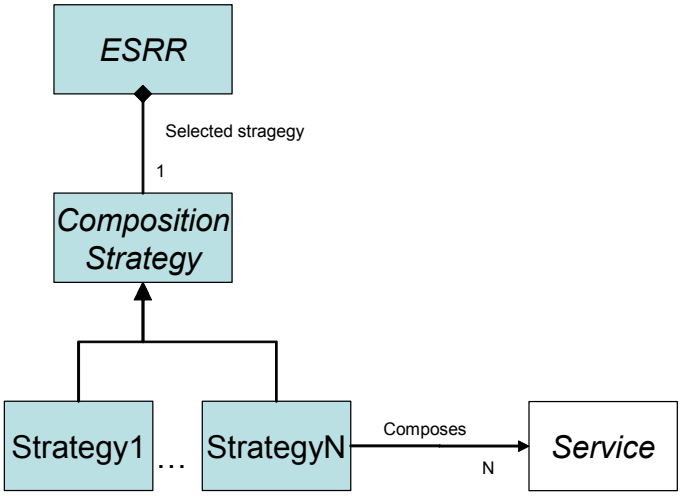

Figure 4-16: Strategy pattern applied to ESRR

In future iterations, an ESRR may be composite, i.e., it can define a new reasoning activating a new ESRR as its composition strategy (see Figure 4-17). This allows keeping the conditions of rules simple and provides support for hierarchical reasoning.

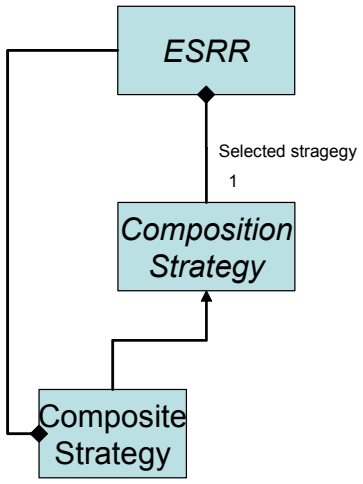

Figure 4-17: Composite pattern applied to ESRR

Figure 4-18 shows the resulting conceptual information structure of ESRR. The idea is that this same logical architecture can potentially be implemented as Java classes, SWRL rules or XML, depending on the software platform available on an Amigo device.

During the execution of an ESRR, its composition rules are checked, and only if the condition of a rule fails, the next rule is checked. Usually, the order of composition rules is such that, in the condition checking, the most specific situation is executed first; rules and more generic solutions are proposed by the latter rules if the former fail. A default strategy has to be provided, for which the rule consists only of an action part that is executed in case all the other rules fail.

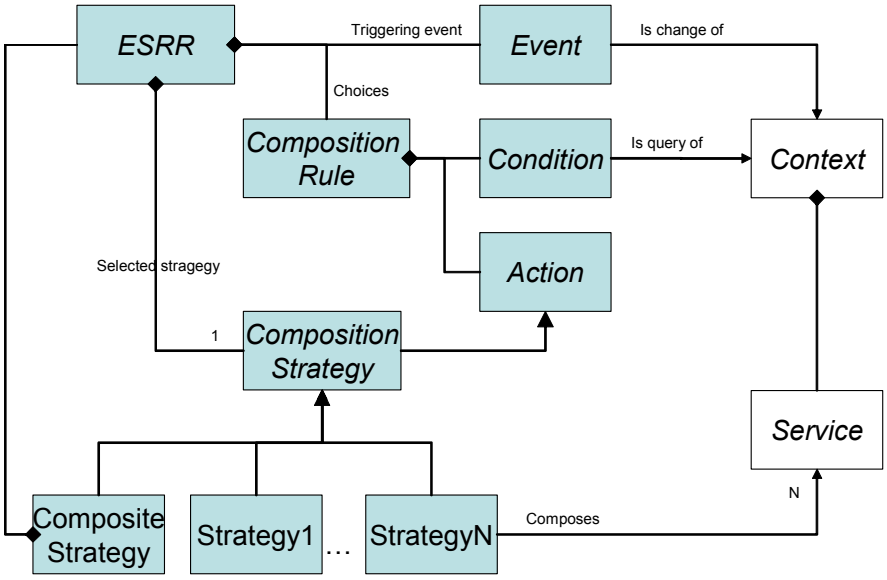

Figure 4-18: Conceptual structure of ESRR

As an example of how ESRR can be used, we consider a simplified situation of adaptation to the available services providing a HomeAVControl capability or one or more of its sub-capabilities in the Amigo home. If we can discover a service providing this capability, we

should use it. Otherwise, we search for services that provide separate MediaServer and MediaRenderer capabilities, and can be composed into one. If that also fails, we may have some special device-specific implementation that can be used. If everything else fails, as default we may use a HomeAVControl capability implementation in the form of an application-internal service that provides minimal functionality required by the application. The ESRR for this example can be presented informally as:

ESRR: Intelligent use of available media servers has the following choices

Rule1: cond(Service1? <HomeAVControl, exact>)

action (“Composition using HomeAVControl capabilities provided by the Service1”)

Rule2: cond(Service1? < MediaServer, exact>, Service2? <MediaRenderer, exact>)

action(“Composition using services Service1 as MediaServer and Service2 as MediaRenderer”)

Rule3: cond(Service1? “Philips37PF7320A_LCD_HDtv “)

action(“Use a special implementation using special capabilities of a service provided this device and internal implementation of supporting services”)

Default:

action(“Use an internal implementation of HomeAVControl service.”)

The action parts specified here in comment text may be implemented, for example, as Java code that uses the discovered services, or as composite service descriptions using the Amigo-S language. We have simplified the situation so that multiple matches are not considered (we assume here that a “best match” is selected). Handling multiple matches can be performed with hierarchical reasoning using composite ESRR.

ESRR is initially provided as a Java application framework that provides the required base classes. The application developer can use these classes to construct the reasoning object structure that is needed for the composition problem. In later iterations, a wizard will be provided that will guide the developer to construct implementation-independent ESRR rules and export the structure so that it can be used by an application in suitable format (Java, SWRL, XML, etc.).

4.4.3 A domain specific language for event-driven service composition

A scripting language for event-driven service composition will provide a means to help developers to integrate and configure devices and services available in the Amigo home in a transparent and useful way to achieve a safer and swifter programming process. There are plenty of examples that can already be found in the scope of Amigo (namely in WP5, WP6 and WP7 applications). These examples demonstrate the necessity of a scripting language dedicated to event-driven service composition. As an illustration, the WP4 Awareness and Notification Service (AWS) notifies the WP6 Monitoring Manager, and a warning is displayed on the home entrance screen. In the same way, when the washing machine finishes its work, Amigo as a housekeeper notifies Maria on the TV or by switching the lamp in the kitchen, depending on Maria’s location and activity (WP5 Clothes Manager). Amigo is also able to recognize visitors, notify by some means (e.g., voice or message on the TV) and open the entrance door (WP5 Entrance Manager). In the scenes above, different simple services and technologies are integrated in complex composite services. Some services are provided by the

intermediary of the Amigo Intelligent User Services (e.g., the Awareness and Notification Service, ANS). Others are discovered by using the Amigo Middleware (e.g., displays, lights).

One approach consists in describing a functional composite application in the form of an abstract workflow that can be executed by integrating on the fly services that are available in the home environment. Our approach relies on introducing a programming language dedicated to the domain of ubiquitous computing. This language enables a user to express the rules that describe the sequence of actions to be performed when an event occurs. We are designing and developing a language dedicated to the domain of event-driven service composition to describe scenarios of ambient intelligence. Domain-specific languages (DSLs) have been successfully used in various application areas and have shown their benefits in terms of accessibility to domain experts, conciseness, readability, safety and robustness.

Our DSL will offer dedicated abstractions and notations to the domain experts that rely on Amigo ontologies (i.e., Amigo-S). This language will permit to use simple services as well as Amigo components (e.g., ANS) to build context-aware service composition.

This scripting language will allow developers to build applications by composing remote functionalities in an event-driven way. Therefore, tools must be provided to developers in order to specify their requirements towards these functionalities. These requirements will be expressed using the Amigo-S language. The service discovery will be processed by the Amigo Service discovery component. Our approach will also rely on the service functionalities' orchestration and composition already supplied by SD-SDCAE. These reuses will be simplified by the use of a compiler able to map our scripting language to a targeted BPEL-based language, which will most probably be used in SD-SDCAE for service execution. This compiler will also ensure the robustness and the compliancy of scripts towards Amigo requirements. This scripting language will use ANS. ANS rules will be expressed in the scripting language to receive contextual information. The WS-Eventing mechanism supported by the Amigo middleware (see Chapter 2) will enrich the targeted language to deal with ANS rules subscription and notification. This scripting language will benefit from the compilation step by performing static checking and thus ensure the robustness of the application at runtime.

An example of such script is shown in Figure 4-19. The Reminder Application subscribes to a change of context by specifying an event-condition-action (ECA) rule (line 11). This rule is specified according to the ANS requirements, and its correctness is statically checked by the compiler. The Proto statement represents a requested service (line 4 and 10). The requested service can be a simple one, such as a display, or an Amigo service such as the ANS. It specifies the service type (e.g., Display, line 4) and refines the service definition using properties (e.g., location, line 6). Some kinds of proto imply to implement some statements: for instance, the Rule proto statement must implement the *upon*, *when* and *do* statements (line 11, 14, and 17). This Rule proto statement enables subscribing to the ANS and specifying what actions have to be taken when receiving the corresponding notification. In our example, the previously defined service Display is used to display a message.

```
1 Application Reminder {
2 {
3   Display display1;
4 }
5 { // ----- Resource Declarations -----
6
7   proto Display display { // Display is the “serviceType”
8     filter { // set service properties
9       location = home_entrance;
10    }
11  }
12
13  proto Rule reminderRule {
14    upon {
15      TrueToFalse(isInHouse(Jerry, SmithHouse));
16    }
17    when {
18      isInHouse(lunchbox, SmithHouse);
19    }
20    do { // warn jerry
21      display1 = display.getService();
22      display1.message(“Jerry, don’t forget your lunchbox!”);
23    }
24  }
25
26
27 }
28 { // ------- Main Section --------
29  activate {
30    subscribe reminderRule;
31  }
32
33  deactivate {
34    unsubscribe reminderRule;
35  }
36 }
```

Figure 4-19: A script example

5 Interoperable service discovery & interaction middleware

Provider

INRIA

Introduction

The role of the interoperable service discovery & interaction (SD&I) middleware is to identify the discovery and interaction middleware protocols that execute on the network and to translate the incoming/outgoing messages of one protocol into messages of another, target protocol. The system parses the incoming/outgoing message and, after having interpreted the semantics of the message, it generates a list of semantic events and uses this list to reconstruct a message for the target protocol, matching the semantics of the original message. The interoperable SD&I middleware acts in a transparent way with regard to discovery and interaction middleware protocols executing and services running on top of them. The supported service discovery protocols are UPnP, SLP and WS-Discovery, while the supported service interaction protocols are SOAP and RMI.

Development status

The final version of the interoperable SD&I middleware is available since M24.

Intended audience

System developers that seek to integrate heterogeneous middleware platforms and their supported service-oriented architectures within dynamic environments.

License

The interoperable SD&I middleware is available under the LGPL license terms.

Language

C

Environment (set-up) info needed if you want to run this sw (service)

The interoperable SD&I middleware does not require any additional software.

Platform

Linux

Tools

None

Files

Source code files are currently available on [Amigo-OSS-SCM] under the *mdwcore/sdi_sii* structure. They will also be made available on [Amigo-OSS-Pub].

Documents

The developer's and user's guides are available on [Amigo-OSS-Pub].

Tasks

None

Bugs

None so far

Patches

None so far

6 Domotic infrastructure

6.1 Overview

The Amigo Domotic Infrastructure aims at presenting heterogeneous physical hardware devices as unified software services using standard service technologies. Nowadays, there is a great diversity of physical device technologies and protocols. Further, there are a number of service technologies that should be supported within the Amigo system.

Therefore, the purpose of the Amigo Domotic Infrastructure is to enable the integration of different device technologies presenting them by means of software services, but isolating the final users (service clients) from the specific base technologies.

Figure 6-1 depicts the proposed domotic architecture:

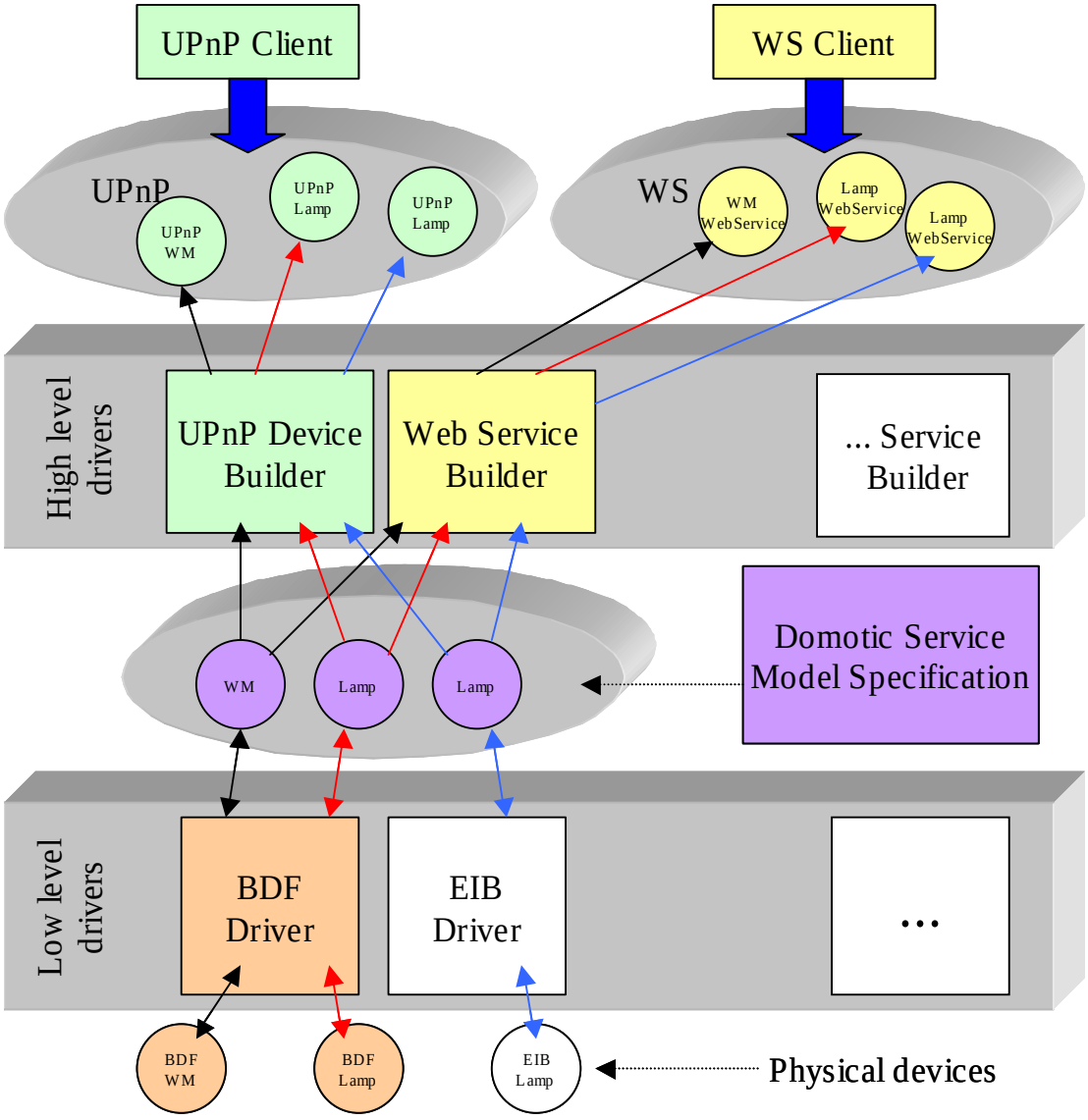

Figure 6-1: The Amigo domotic architecture

The Amigo domotic architecture is based on: extracting the required information about the physical devices by means of drivers to the base technologies like BDF (Fagor Domotic Bus), EIB (European Instalation Bus), EHS (European Home System), X10, etc.; modeling the services using a well-known domotic service specification; and building proxies for the domotic model instances using standard service technologies (UPnP, Web Services, etc.).

The intermediate domotic instances decouple the low-level drivers from the high-level drivers.

Interoperability is achieved by providing several service infrastructures (UPnP, Web Services, etc.) simultaneously to access the same device: for instance, a washing machine or a lamp can be discovered and controlled either using UPnP or a Web Service (WS-Discovery).

The following components have been developed:

- Domotic Service Model Specification
- BDF Driver (Low-Level Driver)
- UPnP Device Builder (High-Level Driver)
- WS Device Builder (High-Level Driver)

6.2 Domotic Service Model

Provider

IKERLAN

Introduction

In order to integrate heterogeneous domotic devices, an abstract description of the available services, not attached to specific domotic technologies, must be specified. This intermediate description is the common element in the domotic proxy generation process.

This component provides any domotic service developer with the abstract reference of the service description.

Development status

Development was finished in M24. Full documentation is in progress.

Intended audience

Low-Level Driver developers must translate and instantiate services from the employed legacy technology to this generic description.

High-Level Driver developers use this reference as a starting point for the high-level proxy generation process.

License

The abstract reference service description provided by the Domotic Service Model will be released under a LGPL license.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Software: .Net for Windows

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

Files are available on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure. They will also be made available on [Amigo-OSS-Pub].

Documents

Documentation (only developer's guide, because it's not a user-oriented component) will be made available on [Amigo-OSS-Pub].

- Developer's Guide: Design principles and UML diagrams

Tasks

None

Bugs

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

Patches

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

6.3 BDF Driver (Low-Level Driver)

Provider

IKERLAN

Introduction

A Low-Level Driver is a base technology-dependent (in this case, BDF) driver that generates and instantiates proxies for the devices that it supports (BDF washing machine, BDF oven, BDF plug, etc.) in a generic (base technology-independent) way.

Development status

Development was finished in M24. Full documentation is in progress.

Intended audience

Low-Level Driver developers. This component is a sample implementation of a Low-Level Driver. New Low-Level Drivers for other domotic base technologies (EIB, X10, etc.) can be developed following the principles described by this module.

License

The software developed will be released under a LGPL license.

The base technology employed (BDF native driver) is under a proprietary license.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware: BDF domotic devices, BDF bridge to RS232, PC/Laptop

OS: Windows XP / Windows Server 2003

Software: .Net for Windows

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

Files are available on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure. They will also be made available on [Amigo-OSS-Pub].

Documents

Documentation (only developer's guide, because it's not a user-oriented component) will be made available on [Amigo-OSS-Pub].

- Developer's Guide: Design principles

Tasks

None

Bugs

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

Patches

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

6.4 UPnP Device Builder (High-Level Driver)

Provider

IKERLAN

Introduction

This High-Level Driver instantiates high-level proxies (UPnP proxies) starting from the generic instances described by the Domotic Service Model component. The proxy instantiation is a dynamic runtime process.

Development status

Development was finished in M24. Full documentation is in progress.

Intended audience

- High-Level Driver developers. This component is a sample implementation of a High-Level driver. New High-Level Drivers (SLP, Jini, etc.) can be developed following the principles described by this module.
- Domotic service clients (UPnP clients).

License

The module will be released under a LGPL license.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware: PC/Laptop

OS: Windows XP / Windows Server 2003

Software: .Net for Windows, Intel UPnP tools

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

Files are available on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure. They will also be made available on [Amigo-OSS-Pub].

Documents

Documentation will be made available on [Amigo-OSS-Pub].

- User's Guide: UPnP device and service description XML files.
- Developer's Guide: Design principles

Tasks

None

Bugs

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

Patches

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

6.5 WS Device Builder (High-Level Driver)

Provider

IKERLAN

Introduction

The goal of the WS Device Builder component is to instantiate high-level proxies (Web services) starting from the generic instances described by the Domotic Service Model component. This component uses the discovery mechanism of the Amigo .Net programming framework (see Section 2.4); thus, the instantiated services can be discovered using WS-Discovery. Services automatically advertise themselves on the network when they are instantiated (WS-Discovery: Hello) and announce their departure (WS-Discovery: Bye) when they are discarded. They also respond to queries (WS-Discovery: Probe and Resolve).

Reflection emit is a run-time feature that allows code to create dynamic assemblies, modules, and types. Instances representing the domotic services – according to the Domotic Service Model component specification – are dynamically created using this feature.

Development status

Development was finished in M24. Full documentation is in progress.

Intended audience

- High-Level Driver developers. This component is a sample implementation of a High-Level driver. New High-Level Drivers (SLP, Jini, etc.) can be developed following the principles described by this module.
- Domotic service clients (Web Service clients).

License

The module will be released under a LGPL license.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware: PC/Laptop

OS: Windows XP / Windows Server 2003

Software: .Net for Windows, Amigo .Net programming framework – discovery mechanism

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

Files are available on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure. They will also be made available on [Amigo-OSS-Pub].

Documents

Documentation will be made available on [Amigo-OSS-Pub].

- User's Guide: WSDL files for domotic device examples
- Developer's Guide: Design principles

Tasks

None

Bugs

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

Patches

None yet, but, if any, they will be reported on [Amigo-OSS-SCM] under the *mdwcore/domotics* structure.

7 Security & privacy

7.1 Security Framework

Provider

Microsoft, IMS

Introduction

This component provides access to the Amigo authentication and authorization service (see Security Services, Section 7.2). It encapsulates the communication and cryptographic primitives that are used for device/user registration, authentication, and authorization with the centralized Amigo security service, which is released as a separate component.

The Amigo security system is based on a centralized Security Service, which may be replicated to achieve higher system reliability. The employed protocol is a simplified Web-service version of Kerberos: shared secrets are established during registration and are subsequently used for mutual authentication. Authorization by the security service is granted following a role-based authorization scheme, and is transmitted securely using encrypted tickets.

The current framework provides convenient abstractions of this underlying protocol, and enables programmers to participate in the security scheme without having to understand the details of the security mechanism. It includes a discovery mechanism that allows automatic fail-over in case of unavailability of a particular instance of the security service, based on WS-Discovery.

Development status

The first prototype version (C# implementation) was distributed to the Amigo partners in M18. Another intermediate version was released in M24.

The Java implementation is still under development; a first prototype version will be made available in M27 on [Amigo-OSS-Pub].

Intended audience

Service and application developers that need to control access to their service/application.

License

C# version: See EMIC license (Annex A).

Java version: The Java libraries will be made available under the LGPL license terms.

Language

C# / Java

Environment (set-up) info needed if you want to run this sw (service)

The security framework will support/employ:

- Hardware: PC/Laptop/PDA/Smartphone
- OS: Windows XP / Windows Server 2003 / Pocket PC 2003 / Smartphone 2003 / Linux
- Software: .Net for Windows / .NetCF for Windows / OSGi / JRE 1.5

Platform

Microsoft .Net 2.0

Microsoft .NetCF 2.0

JVM

Tools

Generic .Net tools, Visual Studio 2005

Eclipse

Files

See [Amigo-OSS-Pub]

Documents

See [Amigo-OSS-Pub]

Tasks

For .Net, there is an intermediate release in M24 and a full release in M30.

The first Java version will be available in M27.

Bugs

None so far

Patches

None so far

7.2 Security Service

Provider

Microsoft

Introduction

The Amigo security system is based on a centralized Security Service, which may be replicated to achieve higher system reliability. The employed protocol is a simplified Web-service version of Kerberos: shared secrets are established during registration and are subsequently used for mutual authentication. Authorization by the security service is granted following a role-based authorization scheme, and is transmitted securely using encrypted tickets.

The role-based authorization scheme works by assigning each registered device/user/service to a specific class, like domotic, admin, mobile, etc. Access to a service of a specific class is granted based on an access matrix, which captures which service class may be used by which device and/or user class.

Development status

The first prototype version was distributed to the Amigo partners in M18 (as a minimal implementation of a security service). Another intermediate version was released in M24.

Intended audience

Service developers as well as application developers that need to control access to their service/application.

License

See EMIC license (Annex A).

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Security services will support/employ:

- Hardware: PC / Laptop
- OS: Windows XP / Windows Server 2003
- Software: .Net for Windows

Platform

Microsoft .Net 2.0

Microsoft .Net 3.0

Tools

Generic .Net tools, Visual Studio 2005

Files

See [Amigo-OSS-Pub]

Documents

See [Amigo-OSS-Pub]

Tasks

Intermediate release in M24 and full release in M30.

Bugs

None so far

Patches

One so far

8 Content distribution

8.1 Introduction

The Content Distribution service (see Figure 8-1) will provide available content in the Amigo home to Amigo services and applications according to the DLNA standard. This is done by gathering available content descriptions (not the actual content to avoid time-consuming and unnecessary copying of content) from UPnP Digital Media Servers (like Windows Media Connect, etc.). Moreover, it has the ability to provide content in a format which suits the renderer's capabilities in the best possible way. For this, copying content might become necessary. Content Distribution will be able to render content to UPnP Digital Media Renderers (DMR). When content is subjected to adaptation some delay might be expected, otherwise rendering will start directly. For other non-UPnP DMRs, like Windows Media Player, etc., an application will need to take care of transferring content to the rendering device via HTTP-GET. The same is applicable for offline consumption.

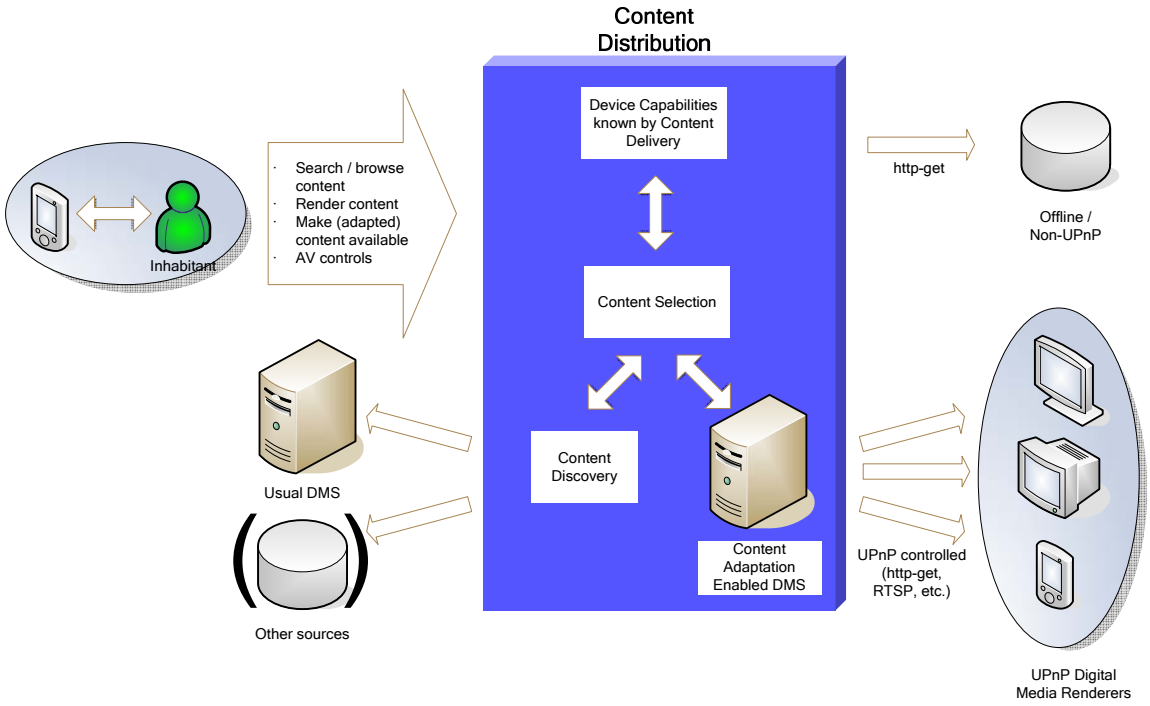

Figure 8-1: Content Distribution in the Amigo home

8.2 Content Distribution Interface

Provider

Microsoft

Development status

Development started in Q1 2006. There was an initial version released in M24. The final version will be released in M30.

Intended audience

Service developers as well as application developers that need (entertaining) content to be rendered or delivered to devices in the Amigo home.

License

See EMIC license (Annex A).

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware: PC/Laptop

OS: Windows XP / Windows Server 2003

Software: .Net for windows

Platform

Microsoft .Net 2.0 runtime.

Microsoft .Net 3.0 runtime.

Tools

None so far

Files

See [Amigo-OSS-Pub]

Documents

Developer's guide: See [Amigo-OSS-Pub]

User's guide: See [Amigo-OSS-Pub]

Tasks

Development started in Q1 2006. Subsequent releases are/will be available as listed below:

- M24: first prototype.
- M30: final prototype.

Bugs

None so far

Patches

None so far

8.3 Content Adaptation Server

Provider

TID

Development status

Development started in Q1 2006. The final prototype will be provided at the end of M30.

- A media server compliant with the UPnP AV DMS template has been developed based on Cyberlink's basic implementation.
- Optional and extended actions have been implemented and created in order to provide a useful server in the content adaptation framework.
- Content Description Interoperability has been implemented providing some reference metadata extraction plugins (for ID3, AVI, MOV and some other format dependent tags) and translating key-value pairs into ontology concepts.
- A semantic registry is provided, enabling RDQL queries to the Content Model.
- Adaptation Interface has been specified. Internal composition at description level has been achieved with some restrictions.

Intended audience

System designers/software maintainers that need to fix bugs/enhance the Content Adaptation subcomponent or developers/integrators of transcoding plugins for Amigo.

License

Content adaptation will be made available under the LGPL license terms.

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

The Content Adaptation subcomponent requires JRE1.5.

Platform

Any system capable of running JRE 1.5

Tools

None so far

Files

CADMS.jar on [Amigo-OSS-SCM].

Documents

Java documentation

Tasks

Development started in Q1 2006. Subsequent releases are/will be available as listed below:

- M24: first prototype.
- M30: final prototype.

Bugs

None so far

Patches

None so far

8.4 Content Discovery

The upcoming generation of new media server devices impels management and control of all the resources that would become available through many different devices.

Every time a new server is connected to the network, the newly shared resources must be centrally controlled and referenced; thus, browsing for a single device is needed in order to have all the media at one's disposal. This handling must be carried out by an engine that is capable of discovering, registering and referencing new resources.

The Content Discovery subsystem (see Figure 8-2) is in charge of discovering new media servers connected to the network and subscribing to their events for possible changes. Once located and identified, the Content Discovery Service will navigate through their contents, exploring and classifying the new available media and referencing them into a central content directory service, which will be aware of all the content resources published on the network.

Provider

TID

Development status

Development started in Q2 2006. The final prototype will be provided at the end of M30.

Discovery of and subscription to UPnP AV DMS's on the network has been implemented, together with: recursive browsing of their contents; and reference creation, update and

deletion in a central content directory service (temporarily located at the Content Adaptation-enabled DMS) implementing UPnP AV ContentDirectory Service with two extended actions.

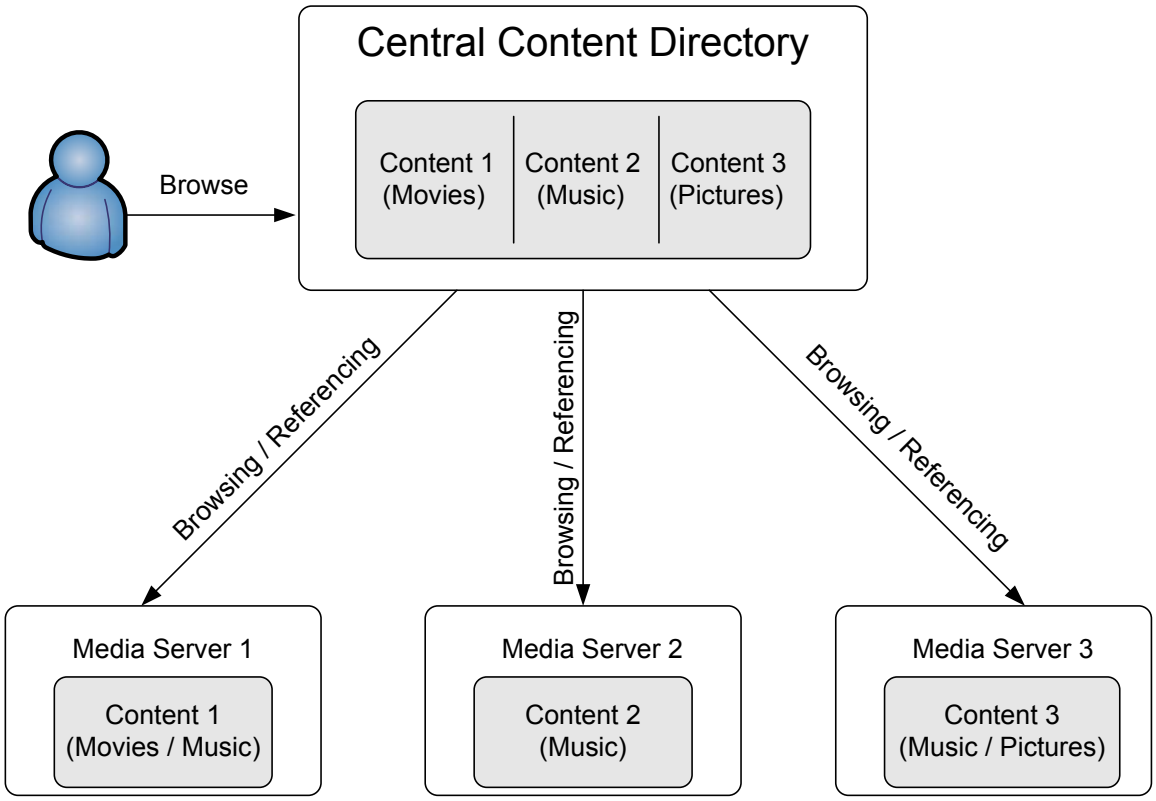

Figure 8-2: Content Discovery browsing and referencing hierarchy

Intended audience

System designers/software maintainers that need to fix bugs/enhance the Content Discovery subcomponent or developers/integrators of transcoding plugins for Amigo.

License

Content adaptation will be made available under the LGPL license terms.

Language

Java

Environment (set-up) info needed in order to run this software (service)

The Content Discovery subcomponent requires JRE1.5 and the OSGi framework.

Platform

Any system capable of running JRE 1.5 and the OSGi framework.

Tools

None so far

Files

amigo-content-discovery.jar on [Amigo-OSS-SCM].

Documents

Java documentation

Tasks

Development has started in Q2 2006. Subsequent releases are/will be available as listed below:

- M24: first prototype.
- M30: final prototype.

Bugs

None so far

Patches

None so far

9 Data store

Provider

Microsoft

Introduction

This component offers a generic storage service to other components and applications inside an Amigo system. There is no restriction on the kind of content that can be stored, and each component or application can open and control access to a sub-store inside the Data Store. It supports also notifications on changes in a sub-store. Data is automatically backed up and restored when necessary.

The Data Store uses a concept of individual compartments that are created on behalf of an owner. The owner of a compartment specifies:

- The structure of each data element;
- The user group and their access rights (planned for future release);
- The events that are generated when elements are modified.

Operations on a compartment include addition, deletion, modification and querying of data elements.

The Data Store is a centralized solution, performing automatic backup and restoration functions when needed to allow a maintenance-free operation.

Development status

There was an initial version released in M24. The final version will be released in M30.

Intended audience

System designers/software maintainers that need to fix bugs/enhance the Data Store component.

License

See EMIC license (Annex A).

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

The Data Store requires the Microsoft .Net 2.0 and 3.0 runtime (http://download.microsoft.com), and the Microsoft SQL database (e.g., Microsoft SQL Express edition).

Platform

Microsoft .Net 2.0 runtime.
Microsoft .Net 3.0 runtime.

Tools

None so far

Files

See [Amigo-OSS-Pub]

Documents

See [Amigo-OSS-Pub]

Tasks

Development started in Q1 2006. Subsequent releases are/will be available as listed below:

- M24: first (intermediate) version.
- M30: final version.

Bugs

None so far

Patches

None so far

10 Accounting & billing

Provider

TID

Introduction

The Accounting and Billing middleware service will provide several functionalities:

- Mediator as defined in the IPDR standard: normalizes to IPDR.
- Data retrieval from (standard and non-standard) metering services (PUSH and PULL).
- Authorized entities may introduce and consult data in a non-standardized way.
- Authorized entities may consult data following the IPDR standard.

Development status

Final prototype at the end of M30. Also: M19 refined architecture, M22 early prototype, M24 first prototype.

Intended audience

Developers of applications requiring Accounting and Billing services.

License

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Software:

- IPDR.org reference libraries

Platform

Any system capable of running JRE 1.5

Tools

None so far

Files

None so far

Documents

None so far

Tasks

Development started in Q2 2006. Subsequent releases are/will be available as listed below:

- M24: first prototype.
- M30: final prototype.

Bugs

None so far

Patches

None so far

11 In-home location management service

Provider

TELIN, FT, PAE, PHI

Introduction

The in-home location management service provides responses to various queries of location-adaptive applications. Inputs of multiple location-determination technologies and services are used and aggregated in order to quickly locate devices or users.

From a WP4 Context Management Service (CMS) perspective, the location management service is a Context Interpreter. Since this is a specialization of a Context Source, this also means that applications can use the location management service as a context source.

The different inputs are context sources to which the location management service is subscribed. By combining the different types of location information, the location determination of users and devices can potentially be more accurate than using a single source of location information. Figure 11-1 shows the high-level architecture of the Location Management Service.

The location management service also simplifies getting location information for location-aware applications and services by providing a single point of contact for location information.

Currently identified as inputs to this service are: an Acoustic Position Estimation Sensor, a raw Bluetooth context source, an RF positioning context source, and an integrated location management system (AmiLoc). Additional context sources for location information can be added if they become available within the Amigo project.

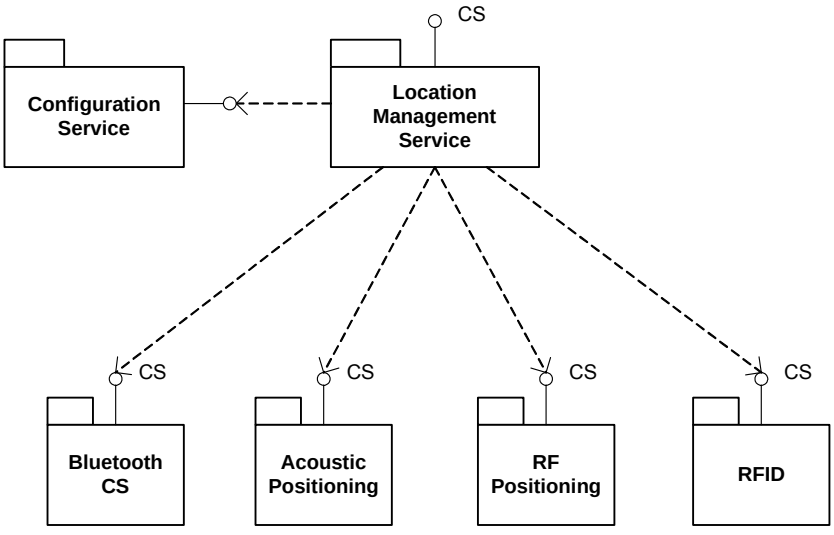

Figure 11-1: Location Management Service high-level architecture.

An example of the information that the location management service provides is given below. This example indicates that a person identified as p.erson@office.company.eu is located in the room known as A2.05@office.company.eu, with a 70% probability. Note that namespace and type information is omitted from this example for clarity.

```
<?xml version="1.0"?>
<rdf:RDF>
<UserLocation>
    <probability>0.7</probability>
    <timestamp>2006-09-18T00:00:00</timestamp>
    <isLocatedIn>
       <Office>
          <floorNumber>2</floorNumber>
          <identifier>A2.05@office.company.eu</identifier>
       </Office>
    </isLocatedIn>
    <isLocationOf>
       <User>
          <identifier>p.erson@office.company.eu</identifier>
       </User>
    </isLocationOf>
</UserLocation>
</rdf:RDF>
```

Development status

Development of the location management service started in Q1 of 2006 with focus on the lower-level Context Sources that have to provide the input for the Location Management Service.

The lower-level Context Sources are in various stages of development, but most are now available from the repository, such as a raw Bluetooth context source, RF positioning, and Acoustic positioning. AmiLoc is available for RDQL queries, and is currently undergoing further work to support SPARQL queries, which is the query language for the CMS.

For the next release, the lower-level Context Sources have to be connected to the location integration component.

Intended audience

Service and application developers that need to locate users and or track the location of users and/or devices.

License

The location management service itself will be released under a LGPL license. The context sources that may provide input to the location management service have different licenses. Note that not all possible inputs are developed within the Amigo project. The known license forms of the different input components are:

- Acoustic Position Estimation Sensor: proprietary license

- RF Positioning: proprietary license

- AmiLoc: proprietary license

- Raw Bluetooth context source: LGPL

- Bluetooth Place Lab : GPL

Language

Java and C#

Environment (set-up) info needed if you want to run this sw (service)

The location management service needs a standard (Java) run-time environment with an application server. Note that the context sources that provide the input may have additional requirements, such as a (set of) microphone array(s) for the Acoustic Position Estimation Sensor.

Platform

Windows/Linux, OSGi

Tools

None yet

Documents

Minimal

Tasks

First release using only AmiLoc as input: M24;

An intermediate release combining more inputs into one location: M27;

The final release of the location management service is scheduled for M30.

Bugs

None known yet

Patches

None yet

12 Conclusion

D3.3 is the second deliverable on the prototype implementation and associated documentation of the Amigo middleware. It comprises the present document and a multitude of other delivered material:

- Developed source code of most Amigo middleware components;
- Developed ontologies in OWL constituting the service description vocabulary and language;
- User's guide and developer's guide documents for components and ontologies; and
- Accompanying Javadoc-style and OWLDoc electronic documentation.

Delivered material besides the present document can be accessed – for the moment, in a restricted way – on the Amigo OSS Repository - Public Web site [Amigo-OSS-Pub]. While we deliver advanced or early implementation versions of most Amigo middleware components, we also report on ongoing conceptual and design work for other middleware components.

D3.3 specifically addresses: the Amigo programming and deployment framework (advanced implementation available); service description vocabulary and language (advanced specification in OWL available); comprehensive service description, discovery, composition, adaptation & execution (conceptual and experimental work, but also early implementations available); interoperable service discovery & interaction middleware (final implementation available); domotic infrastructure (final implementation available); security & privacy (advanced implementation available); content distribution (pretty advanced implementation available); data store (advanced implementation available); accounting & billing (early implementation available); and in-home location management (early implementation available).

Our prototype implementation of the Amigo middleware is currently at a pretty advanced stage, enabling developers of Amigo intelligent user services (WP4) and applications (WP5, WP6, WP7) to already employ most middleware functionalities in their developments. As the application work packages WP5, WP6, WP7 have now pretty much concretized their target application demonstrators, the next step is to start integrating the Amigo middleware components into the development of these demonstrators. At the same time, we continue our work on improving the current versions of Amigo middleware components and on developing new components that reflect our latest conceptual and design work.

Appendix A

MICROSOFT EMIC AMIGO SHARED SOURCE LICENSE FOR NONCOMMERCIAL USE

"The Amigo partners are licensed to use the Deliverable in accordance with the Amigo Consortium Agreement and EU Contract. If and when the Deliverable is released for use by the general public on the terms of the licence below, the Amigo partners (as well as the general public) may also use the Deliverable upon the terms of such licence. However, their use of the Deliverable upon the terms of such licence shall not limit their rights under the Amigo Consortium Agreement or EU Contract."

This License governs use of the accompanying Software (including source code), and your use of the Software constitutes acceptance of this license. **If you do not accept all the terms of this license, you must not use the Software.**

You may use this Software for any non-commercial purpose, subject to the restrictions in this License. Some purposes which can be non-commercial are teaching, academic research, and personal experimentation. You may also distribute this Software with books or other teaching materials, or publish the Software on websites, that are intended to teach the use of the Software.

You may not use or distribute this Software or any derivative works in any form for commercial purposes. Examples of commercial purposes would be running business operations, licensing, leasing, or selling the Software, or distributing the Software for use with commercial products.

You may modify this Software and distribute the modified Software for non-commercial purposes, however, you may not grant rights to the Software or derivative works that are broader than those provided by this License. For example, you may not distribute modifications of the Software under terms that would permit commercial use, or under terms that purport to require the Software or derivative works to be sublicensed to others.

You may use any information in intangible form that you remember after accessing the Software. However, this right does not grant you a license to any of Microsoft's copyrights or patents for anything you might create using such information.

In return, you agree:

1. Not to remove any copyright or other notices from the Software.

2. That if you distribute the Software in source or object form, you will include a verbatim copy of this License.

3. That if you distribute derivative works of the Software in source code form you do so only under a license that includes all of the provisions of this License, and if you distribute derivative works of the Software solely in object form you shall do so only under a license that complies with this License.

4. That if you have modified the Software or created derivative works, and distribute such modifications or derivative works, you will cause the modified files to carry prominent notices so that recipients know that they are not receiving the original Software. Such

notices must state: (i) that you have changed the Software; and (ii) the date of any changes.

5. **THAT THE SOFTWARE COMES "AS IS", WITH NO REPRESENTATIONS, WARRANTIES OR CONDITIONS. THIS MEANS NO EXPRESS, IMPLIED OR STATUTORY REPRESENTATION, WARRANTY OR CONDITION, INCLUDING (WITHOUT LIMITING THE SCOPE OF THIS EXCLUSION), WARRANTIES OR CONDITIONS CONCERNING THE QUALITY OF OR FITNESS FOR ANY PURPOSE OF THE SOFTWARE OR ANY REPRESENTATION OR WARRANTY OF TITLE OR THAT THE USE OF THE SOFTWARE WILL NOT RESULT IN THE INFRINGEMENT OF ANY PERSON'S RIGHTS. ALSO, YOU MUST PASS THIS DISCLAIMER ON WHENEVER YOU DISTRIBUTE THE SOFTWARE OR DERIVATIVE WORKS.**

6. **THAT NEITHER MICROSOFT NOR ANY PERSON OR CORPORATION CONNECETD WITH IT WILL BE LIABLE FOR ANY LOSS OR DAMAGE RELATED TO THE SOFTWARE OR THIS LICENSE. THIS MEANS NO LIABILITY FOR ANY DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL OR INCIDENTAL LOSS OR DAMAGE, NO MATTER WHAT LEGAL THEORY IT IS BASED ON, TO THE MAXIMUM EXTENT THE LAW PERMITSTHIS EXCLUSION. ALSO, YOU MUST PASS THIS LIMITATION OF LIABILITY ON WHENEVER YOU DISTRIBUTE THE SOFTWARE OR DERIVATIVE WORKS.**

7. **THAT THE EXCLUSIONS IN PARAGRAPHS 5 AND 6 ABOVE ARE REASONABLE IN THE CIRCUMSTANCES. IN PARTICULAR, YOU ACKNOWLEDGE (1) THAT THIS SOFTWARE HAS BEEN MADE AVAILABLE TO YOU FREE OF CHARGE, (2) THAT THIS SOFTWARE IS NOT "PRODUCT" QUALITY, BUT HAS BEEN PRODUCED BY A RESEARCH GROUP WHO DESIRE TO MAKE THIS SOFTWARE FREELY AVAILABLE TO PEOPLE WHO WISH TO USE IT FOR NONCOMMERCIAL PURPOSES ONLY, AND (3) THAT BECAUSE THIS SOFTWARE IS NOT OF "PRODUCT" QUALITY (BUT IS THE RESULT OF BASIC RESEARCH), IT IS INEVITABLE THAT THERE WILL BE BUGS AND ERRORS, AND POSSIBLY MORE SERIOUS FAULTS, IN THIS SOFTWARE.**

8. That no technical support will be provided in relation to the Software.

9. That if you sue anyone over patents that you think may apply to the Software or anyone's use of the Software, your license to use the Software under the terms of this License shall end automatically.

10. That your rights under this License shall end automatically if you breach it in any way.

11. That Microsoft reserves all rights not expressly granted to you in this License.

12. That, except to the extent that local laws necessarily apply, this license shall be governed and construed in all respects in accordance with the laws of England and Wales.

References

[Amigo-D2.1] Amigo Consortium. Deliverable D2.1: Specification of the Amigo Abstract Middleware Architecture. April 2005.

[Amigo-D3.2] Amigo Consortium. Deliverable D3.2: Amigo Middleware Core - Prototype Implementation & Documentation. March 2006.

[Amigo-D9.5] Amigo Consortium. Deliverable D9.5: Web site for sharing open source software developed within Amigo. March 2006.

[Amigo-OSS-Pub] Amigo Consortium. Amigo OSS Repository - Public Web Site. http://amigo.gforge.inria.fr/home/index.html

[Amigo-OSS-SCM] Amigo Consortium. Amigo OSS Repository - Source Code Management (SCM). http://gforge.inria.fr/projects/amigo/

[BGI05] S. Ben Mokhtar, N. Georgantas, and V. Issarny. Ad hoc composition of user tasks in pervasive computing environments. In Proceedings of the 4th Workshop on Software Composition (SC'05), 2005.

[BGI06a] S. Ben Mokhtar, N. Georgantas, V. Issarny. COCOA: COnversation-based Service COmposition in PervAsive Computing Environments. In *Proc. IEEE International Conference on Pervasive Services (ICPS'06)*, Lyon, France, June 2006.

[BGI06b] S. Ben Mokhtar, A., N. Georgantas, V. Issarny. Efficient Semantic Service Discovery in Pervasive Computing Environments**.** In *Proc. ACM/IFIP/USENIX 7th International Middleware Conference*, Melbourne, Australia, November 2006.

[BKGI06] Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valerie Issarny. Towards efficient matching of semantic web service capabilities. In Proceedings of the workshop of Web Services Modeling and Testing (WS-MATE'06), 2006.

[CF03] Ion Constantinescu and Boi Faltings. Efficient matchmaking and directory services. In Proceedings of the IEEE International Conference on Web Intelligence (WI'03), 2003.

[LI05] J. Liu and V. Issarny. Signal strength-based service discovery (s3d) in mobile ad hoc networks. In Proceedings of the IEEE personal indoor mobile radio communication (PMRC'05)), 2005.

[P02] M. Paolucci et al., Semantic matching of Web Services capabilities, In Proceedings of the 1st International Semantic Web Conference (ISWC 2002)

[SPS04] Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. Adding owl-s to uddi, implementation and throughput. In Proceedings of the Workshop on Semantic Web Service and Web Process Composition, 2004.